Coherent Accelerator Interface Architecture 1.0

Version 1.1

Advance

11 August 2015

Note: This document contains information on products in the design, sampling and/or initial production phases of development. This information is subject to change without notice. Verify with your IBM field applications engineer that you have the latest version of this document before finalizing a design.

IBM Systems
294 Route 100, Building SOM4
Somers, NY 10589-3216

The IBM home page can be found at **ibm.com®**.

Version 1.1
11 August 2015

Contents

List of Tables

List of Figures

Revision Log

Each release of this document supersedes all previously released versions. The revision log lists all significant changes made to the document since its initial release. In the rest of the document, change bars in the margin indicate that the adjacent text was significantly modified from the previous release of this document.

Revision Date	Version	Contents of Modification
11 August 2015	1.1	CAIA 1.0. • Changed PSL_SPAP_An[v] to PSL_SPAP_An[V] throughout the document. • Changed PSL_CNTL_An to PSL_SCNTL_An throughout the document. • Changed the statement “Access to these registers should be privileged and must be written using a single 64-bit store operation” to “Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.” (for the case where there is one register per PSL slice) throughout the document. • Changed the statement “Access to this register should be privileged and must be written using a single 64-bit store operation” to “Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.” (for the case where there is one register for the whole PSL) throughout the document. • Corrected grammar (removed “is”) in *Section 1 Introduction to Coherent Accelerator Interface Architecture* on page 17. • Added a paragraph to *Section 1 Introduction to Coherent Accelerator Interface Architecture* on page 17. • Changed “address” to “addresses” in *Section 1.1 Organization of a CAIA-Compliant Accelerator* on page 17. • Changed “should” to “must” in the implementation note in *Section 1.3 Reserved Fields and Registers* on page 20. • Added *Section 1.5 Conformance to the CAIA* on page 21. • Defined partition with a footnote in *Section 2 Programming Models* on page 23. • Changed WEQP to SPAP, WE Fetch to PE Fetch, “AM[O]R” to “AMOR”, centered “Physical Address”, and added a legend in the PSL slice block in *Figure 2-1 Accelerator Invocation Process in the Dedicated Process Model* on page 26. • Changed “AMR” to “AMOR” in the Host Procssor block (hypervisor process element) and changed “WEQP” to “SPAP”, “AM[O]R” to “AMOR”, and added a legend in the PSL slice block in *Figure 2-2 Accelerator Invocation Process in the Shared Model* on page 30. • Revised the Scheduled Processes Area in *Figure 2-3 Structure for Scheduled Processes* on page 31 to match the corresponding entries in *Table 2-1 Structure of Scheduled Processes Area* on page 31. • Added a reserved area in *Table 2-1 Structure of Scheduled Processes Area* on page 31. • Added a paragraph, changed storage to memory, changed PSLs to first PSL, and defined process element link in *Section 2.4 Process Management* on page 36. • Changed the heading “Operations Performed by the First PSL (PSL_ID[L,F] = ‘01’)” to “Operations Performed by the First PSL (PSL_ID[F] = ‘1’)” in *Section 2.4.1.2 PSL Procedure for the Time-Sliced Programming Models* on page 38. • Changed the heading “Operations Performed by the Last PSL (PSL_ID[L,F] = ‘00’)” to “Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)” in *Section 2.4.3.2 PSL Procedure for Time-Sliced Programming Models* on page 42. • Changed the headings “Operations Performed by the Last PSL” to “Operations Performed by the Next PSL” and “Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)” to “Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)” in *Section 2.4.3.3 PSL Procedure for AFU-Directed Programming Models* on page 44. • Changed SLB to TLB in *Section 2.4.4.1 Software Procedure* on page 46. • Changed the heading “Operations Performed by the First PSL (PSL_ID[L,F] = ‘x1’)” to “Operations Performed by the First PSL (PSL_ID[F] = ‘1’)” in *Section 2.4.4.2 PSL Procedure for Time-Sliced Programming Models* on page 46 • Changed the headings “Operations Performed by the First PSL (PSL_ID[L,F] = ‘x1’)” to “Operations Performed by the First PSL (PSL_ID[L,F] = ‘01’)” and “Operations Performed by the Last PSL (PSL_ID[L,F] = ‘00’)” to “Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)” in *Section 2.4.4.3 PSL Procedure for AFU-Directed Programming Models* on page 47.

Revision Date	Version	Contents of Modification
11 August 2015 (continued)	1.1	CAIA 1.0 version 1.1 changes (continued) • Changed the heading “Operations Performed by the Last PSL (PSL_ID[L,F] = ‘00’)” to “Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)” in *Section 2.4.5.2 PSL Procedure for Time-Sliced Programming Models* on page 48. • Changed the heading “Operations Performed by the Last PSL (PSL_ID[L,F] = ‘00’)” to “Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)” in *Section 2.4.5.3 PSL Procedure for AFU-Directed Programming Models* on page 50. • Changed the heading “Operations Performed by the Last PSL (PSL_ID[L,F] = ‘00’)” to “Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)” in *Section 2.4.7.2 PSL Procedure for Time-Sliced and AFU-Directed Programming Models* on page 54. • Changed hypervisor to operating system and removed the Process Complete row in *Table 6-1 Interrupt Types* on page 64. • Corrected grammar (removed “for”) and adjusted formatting in the register diagram in *Section 8.1.3 PSL AFU Memory Base Address Register (PSL_AMBAR_An)* on page 78. • Formatting adjustment made in the Warn_OS description in *Section 8.1.16 PSL Context Swap Time Register (PSL_CtxTime)* on page 97. • Changed PSL_CNTL to PSL_Control in *Section 8.1.20 PSL Control Register (PSL_Control)* on page 102. • Changed AFU_CNTL_An[R] to AFU_CNTL_An[RA] in *Section 8.1.21.1 AFU Download Procedure* on page 105. • Formatting adjustment (centered) made to the register diagram “Reserved” field in *Section 8.2.7 PSL Data Storage Interrupt Status Register (PSL_DSISR_An)* on page 133. • Changed the bit name from “R” to “Reserved” in the register diagram in *Section 8.2.13 AFU Control Register (AFU_Cntl_An)* on page 140. • Added the Paged Resolution Handle and the Paged Resolution EA Registers to *Table 9-1 AFU Descriptor* on page 148. • Revised the bit description (bit 48) for the req_prog_model in *Table 9-1 AFU Descriptor* on page 148. • Added *Section 9.1.1 Paged-Response Resolution* on page 150. • Removed “optional” from AFU Descriptor Offset and AFU Descriptor Size in *Table 10-3 VSEC Format* on page 160. • Formatting adjustment made to the AFU Descriptor Offset, revised the description for the PSL Programming Port and the PSL Programming Control to include that the register can also be used to download AFUs, and bits (18:21) descriptions were updated with a 3-bit status for the PSL Programming Control in *Table 10-4 VSEC Description* on page 160. • Added “(Data Only Port of Dual-Bus CAIA Compliant Device)” to the table title in *Table 10-5 PCIe Type 0 Configuration Header (Data Only Port of Dual-Bus CAIA-Compliant Device)* on page 167. • Added the AFU Descriptor to the bulleted list in *Appendix A Memory Maps* on page 171. • Reordered *Appendix A.4 AFU Descriptor Memory Map* on page 178 and *Appendix A.3 PSL Slice Privilege 2 Memory Map* on page 176. • Removed the reserved addresses (x‘30’ - ‘2F’) in *Table A-3. PSL Slice Privilege 1 Memory Map* on page 175. • Changed “Memory Management and Lookaside Buffer Management Registers” to “Memory Management Registers” in *Table A-3. PSL Slice Privilege 1 Memory Map* on page 175. • Updated AFU_CR_len address offset and added addresses x‘22’ - x‘27’ as reserved in *Table A-5. AFU Descriptor Memory Map* on page 178. • Added head_pointer, tail_pointer, and psl_chained_command to the Scheduled Processes Area in *Figure B-2. Application Job and Completion Queue* on page 183.
09 December 2014	1.0	CAIA 1.0. Initial release.

Preface

This document defines the Coherent Accelerator Interface Architecture (CAIA) for the IBM® POWER8® systems. The information contained in this document allows various CAIA-compliant accelerator implementations to meet the needs of a wide variety of systems and applications. Compatibility with the CAIA allows applications and system software to migrate from one implementation to another with minor changes.

For a specific implementation of the CAIA, see the documentation for that accelerator.

Who Should Read This Manual

This manual is intended for designers who plan to develop products that use the CAIA. Readers of this manual should be familiar with the documents listed in *Related Publications* on page 14. In addition, individual implementations of the CAIA should have their own documentation for that implementation.

Document Organization

The CAIA document is divided into two environments: the accelerator function environment (AFE) and the system software environment (SSE). In general, the AFE describes the interface facilities available to an accelerator function. The SSE describes the facilities available to an operating system or to hypervisor software. Together, these two environments define the CAIA. Implementation details and compliance with the architecture for a specific implementation are provided separately.

Document Division	Description
Preface	Describes this document, related documents, the intended audience, and other general information.
Revision Log	Lists all significant changes made to the document since its initial release.
Introduction	Provides a high-level overview of the Coherent Accelerator Interface Architecture (CAIA).
POWER® Service Layer	Describes the additional instructions and facilities, beyond those defined in user-mode environment, that are provided by the CAIA. This section covers instructions and facilities, not available to the application programmer, that affect storage control, interrupts, and timing facilities. The following sections are included: • Privileged Mode Overview • Power ISA (Book III) Compatibility • Context Management • Interrupts • Storage Addressing • PSL Privileged 1 Facilities • PSL Privileged 2 Facilities
Accelerator Function Unit	The facilities in these sections of the document provide the AFUs with the ability to read and write main storage, maintain coherency with the system caches, and perform synchronization primitives.
PCIe Configuration Environment	This section describes the PCIe configuration space for a CAIA-compliant device. The configuration facilities are compatible with the PCIe Generation 3 architecture and allow system software to set the base address, the message-signaled interrupt (MSI-X) capabilities, and various other configuration parameters for the CAIA-compliant device.

Document Division	Description
Appendix	• *Appendix A Memory Maps* maps all the registers defined in the Coherent Accelerator Interface Architecture to the real address space. • *Appendix B Examples* contains examples on how a CAIA-compliant device can be used by applications and system software.
Glossary	Defines terms and acronyms used in this document.

Related Publications

The following documents can be helpful when reading this specification. Contact your IBM representative to obtain any documents that are not available through OpenPOWER Connect.

Power ISA User Instruction Set Architecture - Book I (Version 2.07)

Power ISA Virtual Environment Architecture - Book II (Version 2.07)

Power ISA Operating Environment Architecture (Server Environment) - Book III-S (Version 2.07)

I/O Design Architecture v2 (IODA2) (Version 2.4+)

Coherent Accelerator Processor Interface User's Manual

Conventions and Notation

This document uses standard IBM notation, meaning that bits and bytes are numbered in ascending order from left to right. Thus, for a 4-byte word, bit 0 is the most-significant bit, and bit 31 is the least-significant bit.

MSb																															LSb
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Numbers are generally shown in decimal format, unless designated as follows:

- Hexadecimal values are preceded by an “x” and enclosed in single quotation marks or preceded by “0x”. For example: x‘0A00’ or 0x0A00.
- Binary values in sentences are shown in single quotation marks.
  For example: ‘1010’.

Note: A bit value that is immaterial, which is called a “don't care” bit, is represented by an “X.”

This document uses the following software documentation conventions:

1. Command or instruction names are written in **bold** type. For example: **afu_wr** and **afu_rd**.
2. Field names and variables are written in italic type. Required parameters are enclosed in angle brackets. Optional parameters are enclosed in brackets. For example: **afu*<f,b>_wr[a]***.

This document uses the following symbols:

& Bitwise AND

| Bitwise OR

~ Bitwise NOT

% Modulus

= Equal to

! = Not equal to

≥ Greater than or equal to

≤ Less than or equal to

x >> y Shift to the right; for example, 6 >> 2 = 1; least-significant y bits are dropped

x << y Shift to the left; for example, 3 << 2 = 12; least-significant y bits are replaced zeros

|| Concatenate

References to Registers, Fields, and Bits

Registers are referred to by their full name or by their short name (also called the register mnemonic). Fields are referred to by their field name or by their bit position. The following table describes how registers, fields, and bit ranges are referred to in this document and provides examples.

Table i. Register References

Type of Reference	Format	Example
Reference to a specific register and a specific field using the register short name and the field name	Register_Short_Name[Field_Name]	MSR[R]
Reference to a field using the field name	[Field_Name]	[R]
Reference to a specific register and to multiple fields using the register short name and the field names	Register_Short_Name[Field_Name1, Field_Name2]	MSR[FE0, FE1]
Reference to a specific register and to multiple fields using the register short name and the bit positions.	Register_Short_Name[Bit_Number, Bit_Number]	MSR[52, 55]
Reference to a specific register and to a field using the register short name and the bit position or the bit range.	Register_Short_Name[Bit_Number]	MSR[52]
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]	MSR[39:44]
A field name followed by an equal sign (=) and a value indicates the value for that field.	Register_Short_Name[Field_Name]=*n*	MSR[FE0] = ‘1’ MSR[FE] = x‘1’
	Register_Short_Name[Bit_Number]=*n*	MSR[52] = ‘0’ MSR[52] = x‘0’
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]=*n*	MSR[39:43] = ‘10010’ MSR[39:43] = x‘11’
Where *n* is the binary or hexadecimal value for the field or bits specified in the brackets.		

Endian Order

The *Power ISA* supports both big-endian and little-endian byte-ordering modes. *Book I* of the *Power ISA* describes these modes.

The CAIA supports only big-endian byte ordering. Because the CAIA supports only big-endian byte ordering, the POWER Service Layer (PSL) does *not* implement the optional little-endian byte-ordering mode of the *Power ISA*. The data transfers themselves are simply byte moves, without regard to the numerical significance of any byte. Thus, the big-endian or little-endian issue becomes irrelevant to the actual movement of a block of data. The byte-order mapping only becomes significant when data is fetched or interpreted, for example by an accelerator function.

The bit and byte ordering in the CAIA is big-endian except where noted. The main exception is the PCIe configuration space (*Section 10*). The bit and byte ordering in the PCIe configuration space is little-endian to maintain consistency with the PCIe architecture.

1. Introduction to Coherent Accelerator Interface Architecture

The Coherent Accelerator Interface Architecture (CAIA) defines an accelerator interface structure for coherently attaching accelerators to the IBM Power Systems™ using a standard PCIe bus. The intent is to allow implementation of a wide range of accelerators to optimally address many different market segments.

The CAIA allows a single PSL to support multiple AFUs. Architecturally, the only limit on the number of AFUs is the amount of address space assigned to the PSL. The PSL manages the context and virtual addressing for each AFU. While there is only one physical PSL on each physical CAIA-compliant device, some registers in the PSL are duplicated for each AFU, which essentially provides a "virtual" PSL for each AFU. Providing this virtual PSL allows software to treat each AFU independently.

The CAIA document covers three main areas: the POWER Service Layer (PSL), the accelerator function unit (AFU) interface, and the PCIe configuration environment. The PSL section covers the facilities and procedures provided to system software (hypervisor and operating systems) for managing a CAIA-compliant device. The AFU interface section describes the interface facilities available to an accelerator function. The PCIe configuration environment covers the architecture for the configuration space when a CAIA-compliant device is connected using a PCIe bus.

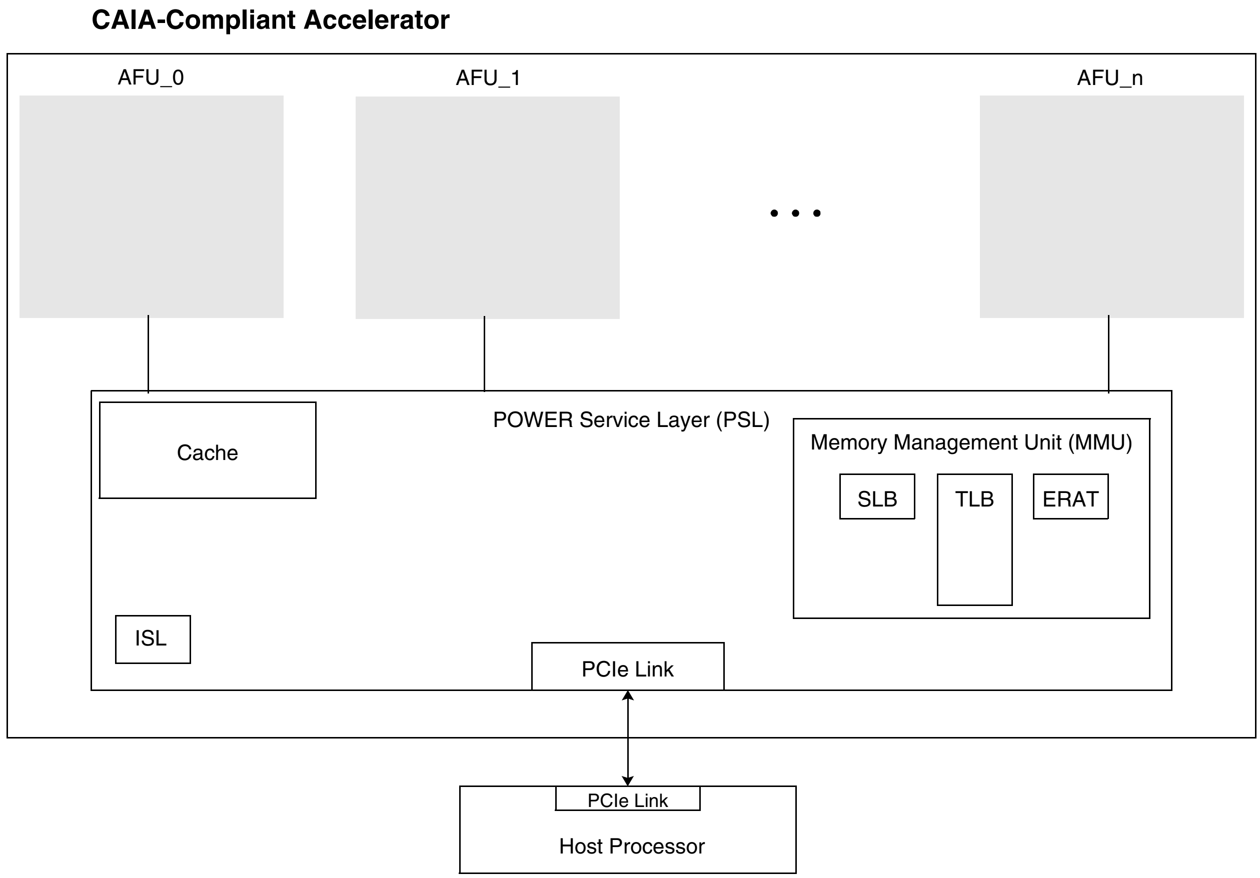
1.1 Organization of a CAIA-Compliant Accelerator

Logically, the CAIA defines two functional components: the PSL and the AFU. The PSL in a CAIA-compliant accelerator provides the interface to the host processor. Effective addresses from an AFU are translated to physical addresses in system memory by the PSL. The PSL also provides miscellaneous management for the AFUs. Although the CAIA architecture defines interfaces for up to four AFUs per PSL, early implementations support only a single AFU. The AFU can be dedicated to a single application or shared between multiple applications. However, only the dedicated programming model is currently supported.

Physically, a CAIA-compliant accelerator can consist of a single chip, a multi-chip module (or modules), or multiple single-chip modules on a system board or other second-level package. The design depends on the technology used, and on the cost and performance characteristics of the intended design point.

Figure 1-1 on page 18 illustrates a CAIA-compliant accelerator with several (n) AFUs connected to the PSL. All the AFUs share a single cache.

Figure 1-1. CAIA-Compliant Processor System



AFU	Accelerator Function Unit	MMU	Memory Management Unit
PSL	POWER Service Layer	SLB	Segment Lookaside Buffer
ISL	Interrupt Source Layer	TLB	Translation Lookaside Buffer
Cache	Cache for data accessed by AFUs	ERAT	Effective-to-Real Address Translation

1.1.1 POWER Service Layer

A CAIA-compliant processor includes a POWER Service Layer. The PSL is the bridge to the system for the AFU, and it provides the address translation and system memory cache for the AFUs. In addition, the PSL provides miscellaneous facilities for the host processor to manage the virtualization of the AFUs, interrupts, and memory management.

The PSL consist of several functional units (such as the memory-protection tables). Hardware resources defined in the CAIA are mapped explicitly to the real address space seen by the host processor. Therefore, any host processor can address any of these resources directly by using an appropriate effective address value. A primary function of the PSL is the physical separation of the AFUs so they appear to the system as independent units.

1.1.2 Accelerator Function Unit

Note: The AFU functional definition is outside the scope of the CAIA.

A CAIA-compliant processor includes one or more AFUs. The AFUs are user-defined functions for accelerating applications. They typically process data and initiate any required data transfers to perform their allocated tasks.

The purpose of an AFU is to provide applications with a higher computational unit density for hardware acceleration of functions to improve the performance of the application and off-load the host processor. Using an AFU for application acceleration allows for cost-effective processing over a wide range of applications.

When an application requests use of an AFU, a process element is added to the process-element linked list describing the application's process state. The process element also contains a work element descriptor (WED) provided by the application. The WED can contain the full description of the job to be performed or a pointer to other main memory structures in the application's memory space. Several programming models are described providing for an AFU to be used by any application or for an AFU to be dedicated to a single application.

1.2 Main Storage Addressing

The addressing of main storage in the CAIA is compatible with the addressing defined in the Power ISA. The CAIA builds upon the concepts of the Power ISA and extends the addressing of main storage by the AFU.

The AFU uses an effective address to access main storage. The effective address is computed by the AFU and is provided to the PSL. The effective address is translated to a real address according to the procedures described in the overview of address translation in Power ISA, Book III. The real address is the location in main storage that is referenced by the translated effective address.

All the AFUs share main storage with the host processor. All information held in this level of storage is visible to all processors and devices in the system. This storage area can either be uniform in structure or can be part of a hierarchical cache structure. Programs reference this level of storage by using an effective address.

1.2.1 Main Storage Attributes

The main storage of a system typically includes both general-purpose and nonvolatile storage. It also includes special-purpose hardware registers or arrays used for functions such as system configuration, data-transfer synchronization, memory-mapped I/O, and I/O subsystems.

Table 1-1 lists the sizes of address spaces in main storage.

Table 1-1. Sizes of Main Storage Address Spaces (Page 1 of 2)

Address Space	Size	Description
Real Address Space	2^m bytes	where $m \leq 60$
Effective Address Space	2^{64} bytes	An effective address is translated to a virtual address using the segment lookaside buffer (SLB).
Virtual Address Space	2^n bytes	where $65 \leq n \leq 78$ A virtual address is translated to a real address using the page table.
Note: The values of "m," "n," and "p" are implementation dependent.		

Table 1-1. Sizes of Main Storage Address Spaces (Page 2 of 2)

Address Space	Size	Description
Real Page (Base)	2^{12} bytes	
Virtual Page	2^p bytes	where $12 \leq p \leq 28$ Up to eight page sizes can be supported simultaneously. A small 4 KB (p = 12) page is always supported. The number of large pages and their sizes are implementation dependent.
Segment Size	2^s bytes	where s = 28 or 40 The number of virtual segments is $2^{(n - s)}$ where $65 \leq n \leq 78$
Note: The values of “m,” “n,” and “p” are implementation dependent.		

1.3 Reserved Fields and Registers

All reserved fields must be set to zero.

A specific instantiation of the CAIA must handle reserved bits as follows:

- Ignore the reserved bits on writes and return zeros for the reserved bits on reads; or
- All reserved registers must be initialized to zero, unless otherwise indicated.

Software is not required to initialize reserved registers. An implementation must decode all reserved registers for both reads and writes. The handling of reserved register values in a specific instantiation of the CAIA is implementation dependent. For each reserved register, an implementation takes one of the following actions:

- Ignore the value on writes and return zeros for reads of the register.
- Maintain the state of the reserved register.

Fields and registers that are currently defined as reserved might become defined in future versions of the CAIA. If a reserved register or register field is defined in a future version of the CAIA, a state of zero should be defined in a manner that is backwards compatible with previous versions of the CAIA.

Implementation Note:

An implementation of the CAIA must never use reserved fields or registers for an implementation-dependent purpose. All defined, reserved, and implementation-dependent registers must be decoded for both reads and writes. Furthermore, the range of addresses for a given area in the memory map must be a multiple of at least 4 KB (that is, the smallest page addresses). An implementation should consider the range to be a multiple of a larger supported page size to minimize the number of page table entries required to map the address range. See *Appendix A Memory Maps* on page 171 for more details.

1.4 Implementation-Dependent Fields and Registers

The CAIA provides implementation-dependent fields and registers. These fields and registers are intended for implementation-dependent purposes and will not be defined by future versions of the CAIA. Unused fields and registers should be handled in the same manner as reserved fields and registers.

For descriptions of the implementation-dependent fields and registers, see the documentation for the specific implementation.

1.5 Conformance to the CAIA

Any implementation of this architecture must adhere to the following set of numbered conformance clauses to claim conformance to CAIA:

1. Must implement at least one of the specified programming models.
2. Must implement the required facilities defined in this specification.
3. Optionally, may implement optional categories defined in this specification.

Table 1-2. CAIA Conformance (Page 1 of 2)

Register Name	Required/Optional
Per AFU Slice Facilities (Privilege 1)	
PSL State Register (PSL_SR_An)	Required
PSL Logical Partition ID Register (PSL_LPID_An)	Required
PSL AFU Memory Base Address Register (PSL_AMBAR_An)	Optional
PSL AFU Scratch Pad Offset Register (PSL_SPOffset_An)	Optional
PSL ID Register (PSL_ID_An)	Required
PSL Slice Error Register (PSL_SERR_An)	Optional
PSL Storage Description Register (PSL_SDR_An)	Required
PSL Authority Mask Override Register (PSL_AMOR_An)	Required
Hypervisor Accelerator Utilization Record Pointer Register (HAURP_An)	Optional
PSL Scheduled Processes Area Pointer Register (PSL_SPAP_An)	Required
PSL Linked List Command Register (PSL_LLCMD_An)	Optional
PSL Slice Control Register (PSL_SCNTL_An)	Required
PSL Context Swap Time Slice Register (PSL_CtxTime_An)	Optional
PSL IVTE Offset Register (PSL_IVTE_Offset_An)	Required
PSL IVTE Limit Register (PSL_IVTE_Limit_An)	Required
PSL Facilities (Privilege 1)	
PSL Context Swap Time Register (PSL_CtxTime)	Optional
PSL Error Interrupt Register (PSL_ErrIVTE)	Required
PSL Key One Register (PSL_KEY1)	Required
PSL Key Two Register (PSL_KEY2)	Required
PSL Control Register (PSL_Control)	Required

Table 1-2. CAIA Conformance (Page 2 of 2)

Register Name	Required/Optional
AFU Download Control Register (AFU_DLCNTL)	Optional
AFU Download Address Register (AFU_DLADDR)	Optional
PSL Lookaside Buffer Invalidate Selection Register (PSL_LBISEL)	Required
PSL SLB Invalidate Entry Register (PSL_SLBIE)	Required
PSL SLB Invalidate All Register (PSL_SLBIA)	Required
PSL TLB Invalidate Entry (PSL_TLBIE)	Required
PSL TLB Invalidate All (PSL_TLBIA)	Required
PSL AFU Selection Register (PSL_AFUSEL)	Required
Privilege 2	
PSL Process and Thread Identification Register (PSL_PID_TID_An)	Required
Context Save/Restore Pointer Register (CSRP_An)	Optional
Accelerator Utilization Record Pointer Zero Register (AURP0_An)	Optional (deprecated)
Accelerator Utilization Record Pointer One Register (AURP1_An)	Optional (deprecated)
Storage Segment Table Pointer Zero Register (SSTP0_An)	Required
Storage Segment Table Pointer One Register (SSTP1_An)	Required
PSL Authority Mask Register (PSL_AMR_An)	Required
SLB Invalidate Entry Register (SLBIE_An)	Required
SLB Invalidate All Register (SLBIA_An)	Required
SLB Invalidate Selection Register (SLBI_Select_An)	Required
PSL Data Storage Interrupt Status Register (PSL_DSISR_An)	Required
PSL Data Address Register (PSL_DAR_An)	Required
PSL Data Segment Register (PSL_DSR_An)	Required
PSL Translation Fault Control Register (PSL_TFC_An)	Required
PSL Process Element Handle Register (PSL_PEHandle_An)	Optional
PSL Error Status Register (PSL_ErrStat_An)	Required
AFU Control Register (AFU_Cntl_An)	Required
AFU Error Register (AFU_ERR_An)	Required
PSL WED Register (PSL_WED_An)	Required
PCIe Configuration Space	
PCIe Type 0 Configuration Space	Required
CAIA Vendor-Specific Extended Capability Structure	Required
AFU Descriptor for compliant AFUs	Required

2. Programming Models

The Coherent Accelerator Interface Architecture (CAIA) defines several programming models for virtualization of an acceleration function unit (AFU):

- Dedicated-process programming model (no AFU virtualization)
- Shared programming models, which include these two types:
  - PSL-controlled shared programming models (AFU time-sliced virtualization)
  - AFU-directed shared programming models (AFU-controlled process element selection virtualization)

Architecture Note:

The AFU-directed programming model, where the AFU selects a context from the process element linked list to use for a transfer, is intended for the *Networking* and *Storage* market segments. For these types of applications, the required address context is selected based on a packet received from a network or which process is accessing storage. A CAIA-compliant device can also act as system memory or the lowest point of coherency (LPC). In this model, the process element and address translation are not required. The LPC model can also be used in combination with the other programming models but might not be supported by all devices.

In the dedicated process model, the AFU is dedicated to a single application or process under a single operating system (partition[1]). The single application can act as an “Application as a Service” and funnel other application requests to the accelerator, providing virtualization within a partition.

In the PSL-controlled shared and AFU-directed shared programming models, the AFU can be shared by multiple partitions. The shared models require a system hypervisor to virtualize the AFU so that each operating system can access the AFU. For single-partition systems not running a hypervisor, the AFU is owned by the operating system. In both cases, the operating system can virtualize the AFU so that each process or application can access the AFU.

For the AFU-directed shared programming model, the AFU selects a process element using a process handle. The process handle is an implementation-specific value provided to the host process when registering its context with the AFU (that is, calling system software to add the process element to the process element linked list). While the process handle is implementation specific, the lower 16-bits of the process handle must be the offset of the process element within the process element linked list.

The “process element” contains the process state for the corresponding application. The work element descriptor (WED) contained in the process element can be a single job requested by an application or it can contain a pointer to a queue of jobs. In the latter case, the WED is a pointer to the job request queue in the application’s address space.

This document does not cover all aspects of the programming models. The intent of this section is to provide a reference for how the AFUs can be shared by all or a subset of the processes in the system. This section defines the infrastructure for setting up the process state and sending a work element descriptor (WED) to an AFU to start a job in a virtualized environment. The function performed by an AFU is implementation dependent.

1. A partition is a single operating-system image. There can be multiple operating-system images running on a system. They can be of the same type (Linux) or of different types (Linux, AIX, other).

2.1 Dedicated-Process Programming Model

The dedicated-process programming model is implementation specific. *Figure 2-1 Accelerator Invocation Process in the Dedicated Process Model* on page 26 shows how an application invokes an accelerator under the dedicated-process programming model.

In this model, a single process owns the AFU. Because the AFU is dedicated to a single process, the programming model is not defined in this document. For more information, see the documentation for the specific implementation.

Because the AFU is owned by a single process, the hypervisor initializes the PSL for the owning partition and the operating system initializes the PSL for the owning process at the time when the AFU is assigned. The following information is initialized:

The following registers are initialized by the hypervisor:

- PSL Slice Control Register (PSL_SCNTL_An)
- Real Address (RA) Scheduled Processes Area Pointer (PSL_SPAP_An)
- PSL Authority Mask Override Register (PSL_AMOR_An)
- Interrupt Vector Table Entry Offset (PSL_IVTE_Offset_An)
- Interrupt Vector Table Entry Limit (PSL_IVTE_Limit_An)
- PSL State Register (PSL_SR_An)
- PSL Logical Partition ID (PSL_LPID_An)
- Real address (RA) Hypervisor Accelerator Utilization Record Pointer (HAURP_An) *{This pointer should be disabled.}*
- PSL Storage Description Register (PSL_SDR_An)

The following registers are initialized by the operating system:

- PSL Process and Thread Identification (PSL_PID_TID_An)
- Effective Address (EA) Context Save/Restore Pointer (CSRP_An) *{This pointer should be disabled.}*
- Virtual Address (VA) Accelerator Utilization Record Pointer (AURP0_An) and (AURP1_An) *{This pointer should be disabled.}*
- Virtual Address (VA) Storage Segment Table Pointer (SSTP0_An) and (SSTP1_An)
- PSL Authority Mask (PSL_AMR_An)
- PSL Work Element Descriptor (PSL_WED_An)

2.1.1 Starting and Stopping an AFU in the Dedicated-Process Model

In a dedicated-process programming model, an AFU is started and stopped by system software.

The following procedure should be used to start an AFU.

1. System software must initialize the state of the PSL.
   All the required Privileged 1, Privileged 1 Slice, and Privileged 2 Slice registers should be initialized so that the address context for the processes and other contexts such as the interrupt vector table entries can be used.
2. System software must set the AFU Slice Reset bit in the AFU_Cntl_An Register (AFU_Cntl_An[RA]). Setting the AFU Slice Reset starts a reset sequence for the corresponding AFU. Initiating a reset sequence also disables the AFU. The AFU does not respond to the problem state MMIO region while disabled.
3. System software must poll the AFU Slice Reset Status for the AFU Slice Reset Sequence to be complete (AFU_Cntl_An[RS] = ‘10’).

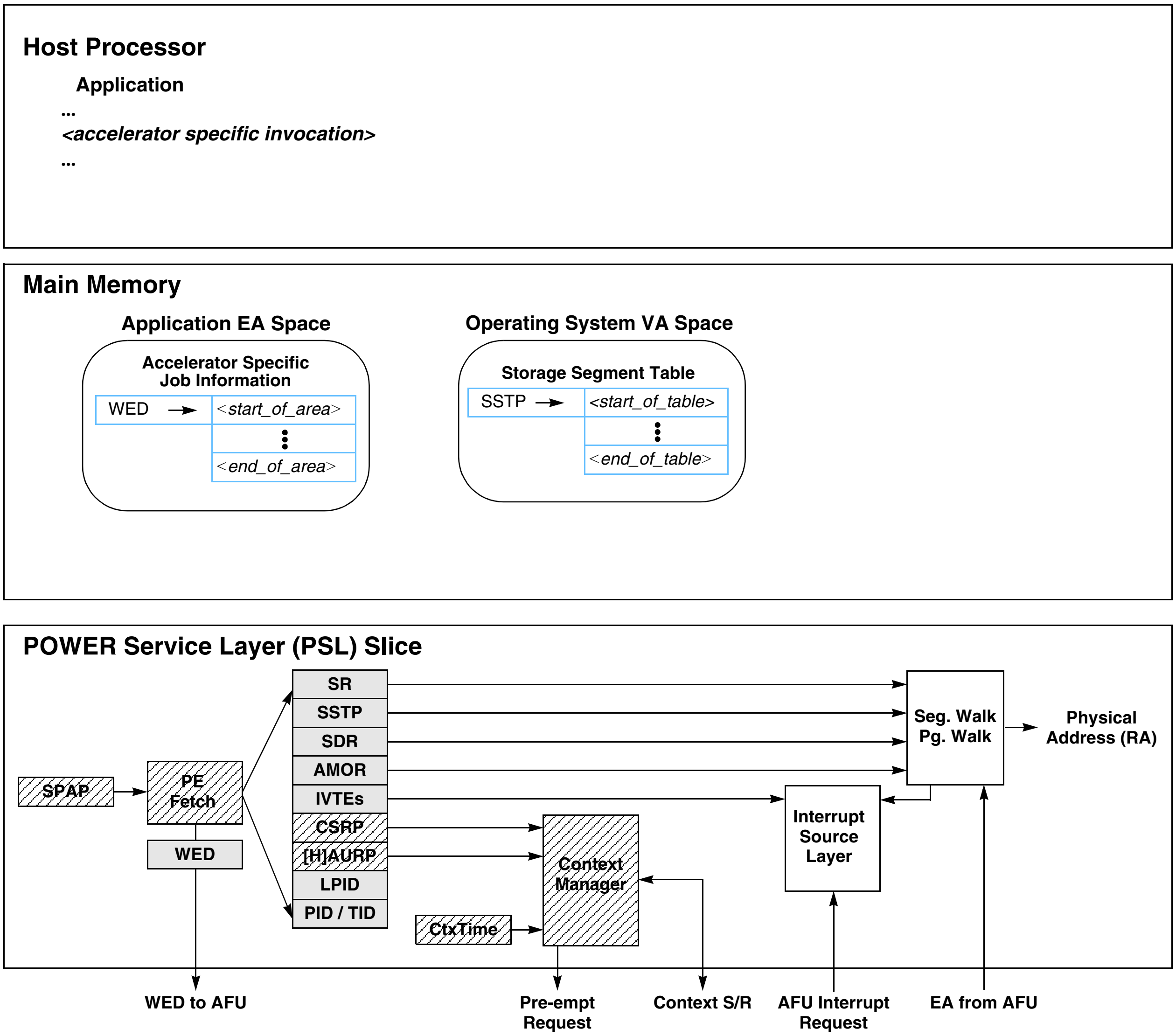
4. System software must set the WED if required by the AFU at start time.
   The WED is initialized by writing a 64-bit WED value to the PSL_WED_An Register.
5. System software must set the AFU Enable bit in the AFU_Cntl_An Register (AFU_Cntl_An[E]).
   The state of the AFU Enable Status must be a ‘00’ before system software can set the AFU Enable bit to a ‘1’ for a start command to be issued to the AFU by the PSL. The WED is passed to the AFU when the start command is issued.
6. System software must poll the AFU Enable Status for the AFU Slice Enabled (AFU_Cntl_An[ES] = ‘10’).
   The AFU_Cntl_An[ES] field is set to ‘10’ when the PSL and AFU are initialized, running, and able to accept MMIO. After the AFU is running, system memory accesses can be performed by the AFU and problem state MMIOs can be performed by software.

   Note: If problem state registers are required to be initialized in the AFU before the application can start, the AFU should provide a mechanism for starting the accelerator and should not depend on the start command issued by the PSL.

The following procedure should be used to stop an AFU.

1. System software must set the AFU Slice Reset bit in the AFU_Cntl_An Register (AFU_Cntl_An[RA]).
   Setting the AFU Slice Reset starts a reset sequence for the corresponding AFU. Initiating a reset sequence also disables the AFU. The AFU does not respond to the problem state MMIO region while disabled.
2. System software must poll the AFU Slice Reset Status for the AFU Slice Reset Sequence to be complete (AFU_Cntl_An[RS] = ‘10’).

Figure 2-1. Accelerator Invocation Process in the Dedicated Process Model



Legend:
SPAP = Scheduled Processes Area Pointer
PE Fetch = Process Element Fetch
Note: Striped blocks are not applicable for this programming model. Shaded blocks are either sourced from the Process Element if the Schedule Process Area Pointer is valid (PSL_SPAP_An[V] = ‘1’) or MMIO registers if the SPAP is not valid (PSL_SPAP_An[V] = ‘0’).

2.2 Shared Programming Models

The shared programming models allow for all or a subset of processes from all or a subset of partitions in the system to use an AFU. There are two programming models where the AFU is shared by multiple processes and partitions; PSL time-sliced shared and AFU-directed shared. *Figure 2-2 Accelerator Invocation Process in the Shared Model* on page 30 shows how an application invokes an AFU under the shared programming model.

In this model, the system hypervisor owns the AFU and makes the function available to all operating systems. For an AFU to support virtualization by the system hypervisor, the AFU must adhere to the following requirements:

- An application's job request must be autonomous (that is, the state does not need to be maintained between jobs),
  -- OR --
  The AFU must provide a context save and restore mechanism.
- An application's job request must be guaranteed by the AFU to complete in a specified amount of time, including any translation faults,
  -- OR --
  The AFU must provide the ability to preempt the processing of the job.
- The AFU must be guaranteed fairness between processes when operating in the AFU-directed shared programming model.

In the case where an AFU can be preempted, the AFU can either require the current job to be restarted from the beginning, or it can provide a method to save and restore the context so that the current job can be restarted from the preemption point at a later time.

For the shared model, the application is required to make an operating-system system call with at least the following information:

- An AFU type (AFU_Type)
  The AFU type describes the targeted acceleration function for the system call. The AFU_Type is a system-specific value.
- A work element descriptor (WED)
  This document does not define the contents of the WED. The WED is AFU implementation specific and can be in the form of an AFU command, an effective address pointer to a user-defined structure, an effective address pointer to a queue of commands, or any other data structure to describe the work to be done by the AFU.
- An Authority Mask Register (AMR) value
  The AMR value is the AMR state to use for the current process. The value passed to the operating system is similar to an application setting the AMR in the processor by using SPR 13 or by calling a system library. If the PSL and AFU implementations do not support a User Authority Mask Override Register (UAMOR), the operating system should apply the current UAMOR value to the AMR value before passing the AMR in the hypervisor call (hcall). The UAMOR is not described in this document. For more information about the UAMOR, see the *Power ISA, Book III*. The hypervisor can optionally apply the current Authority Mask Override Register (AMOR) value before placing the AMR into the process element. The PSL applies the PSL_AMOR_An when updating the PSL_AMR_An Register from the process element.

- A Context Save/Restore Area Pointer (CSRP)
  The CSRP is the effective address of an area in the application's memory space for the AFU to save and restore the context state. This pointer is optional if no state is required to be saved between jobs or when a job is preempted. The context save/restore area must be pinned system memory.

Upon receiving the system call (syscall), the operating system verifies that the application has registered and been given the authority to use the AFU. The operating system then calls the hypervisor (hcall) with at least the following information:

- A work element descriptor (WED)
- An Authority Mask Register (AMR) value, masked with the current PSL_AMOR_An Register value by the PSL and optionally masked with the current UAMOR by the hypervisor.
- An effective address (EA) Context Save/Restore Area Pointer (CSRP)
- A process ID (PID) and optional thread ID (TID)
- A virtual address (VA) Accelerator Utilization Record Pointer (AURP)
- The virtual address of the Storage Segment Table Pointer (SSTP)
- A logical interrupt service number (LISN)

Upon receiving the hypervisor call (hcall), the hypervisor verifies that the operating system has registered and been given the authority to use the AFU. The hypervisor then puts the process element into the process element linked list for the corresponding AFU type. The process element minimally contains the following:

- A work element descriptor (WED)
- An Authority Mask Register (AMR) value, masked with the current AMOR
- An effective address Context Save/Restore Area Pointer (CSRP)
- A process ID (PID) and optional thread ID (TID)
- A virtual address Accelerator Utilization Record Pointer (AURP)
- The virtual address of the Storage Segment Table Pointer (SSTP)
- Interrupt VectorTable (IVTE_Offset_n, IVTE_Range_n), derived from the LISNs in the hypervisor call parameters.
- A State Register (SR) value
- A logical partition ID (LPID)
- A real address (RA) hypervisor accelerator utilization record pointer (HAURP)
- The Storage Descriptor Register (SDR)

The hypervisor initializes the following PSL registers:

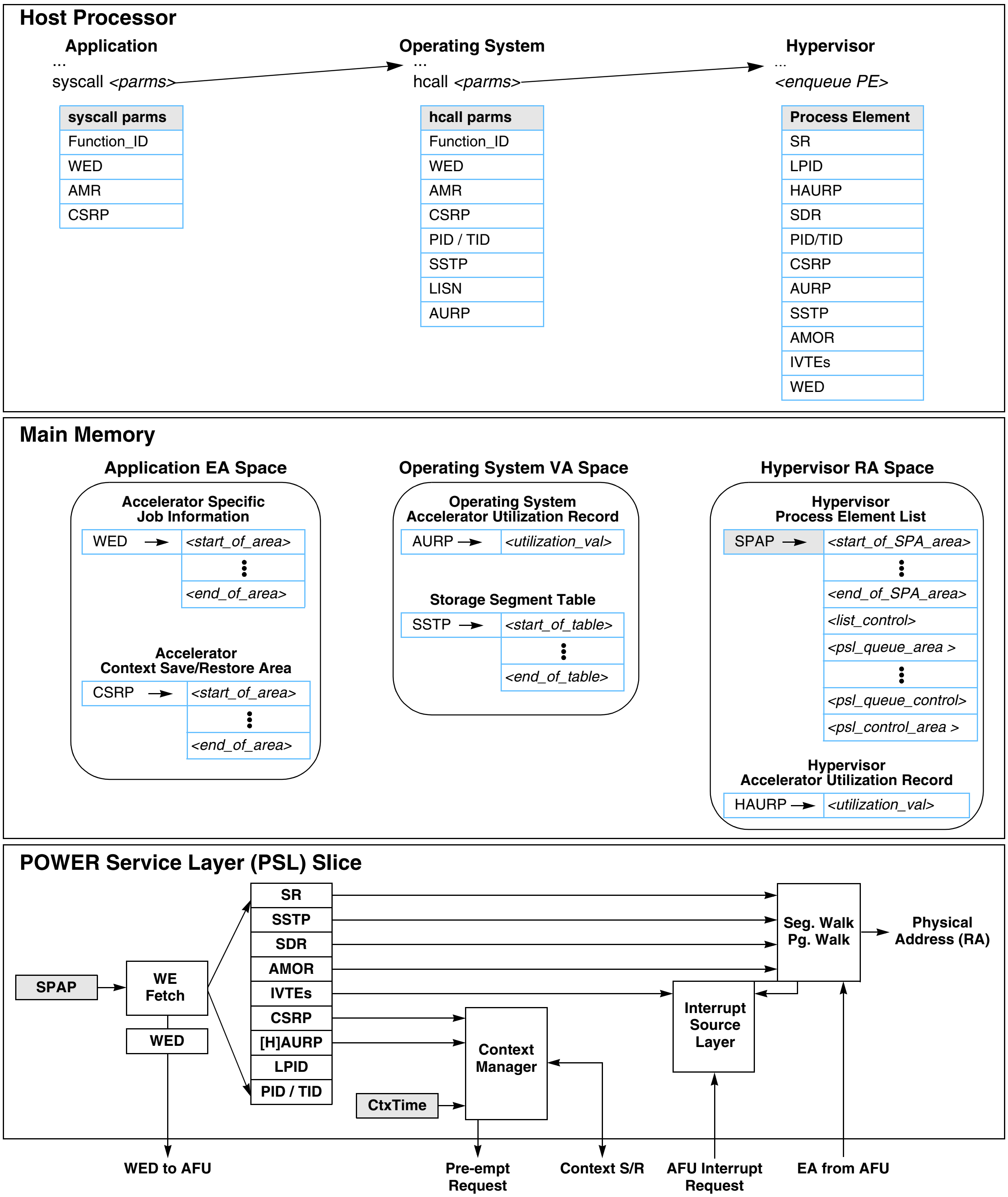
- PSL Control Register (PSL_SCNTL_An)
- Real address Scheduled Processes Area Pointer (PSL_SPAP_An)
- PSL Authority Mask Override Register (PSL_AMOR_An)

2.2.1 Starting and Stopping an AFU in the Shared Models

In the shared, PSL-controlled time-sliced programming model, the AFU is automatically started and stopped by the PSL. The PSL essentially follows the procedures defined in *Section 2.1.1 Starting and Stopping an AFU in the Dedicated-Process Model* on page 24.

In the AFU-directed shared programming model, starting and stopping an AFU process is an AFU implementation-specific procedure.

Figure 2-2. Accelerator Invocation Process in the Shared Model



Legend: Shaded blocks are sourced from the MMIO registers.

2.3 Scheduled Processes Area

In the virtualization programming models, the PSL reads process elements from a structure located in system memory called the scheduled processes area (SPA). The SPA contains a list of processes to be serviced by the AFUs. The process elements contain the address context and other state information for the processes scheduled to run on the AFUs assigned to service the SPA structure. The SPA structure consists of two sections: a linked list maintained by system software and a circular queue maintained by the PSL. The circular queue section is only used for programming models where the context swaps are managed by the PSL. For all other programming models, the circular queue section is not used. *Figure 2-3* shows the structure that contains the processes scheduled for the AFUs.

Figure 2-3. Structure for Scheduled Processes

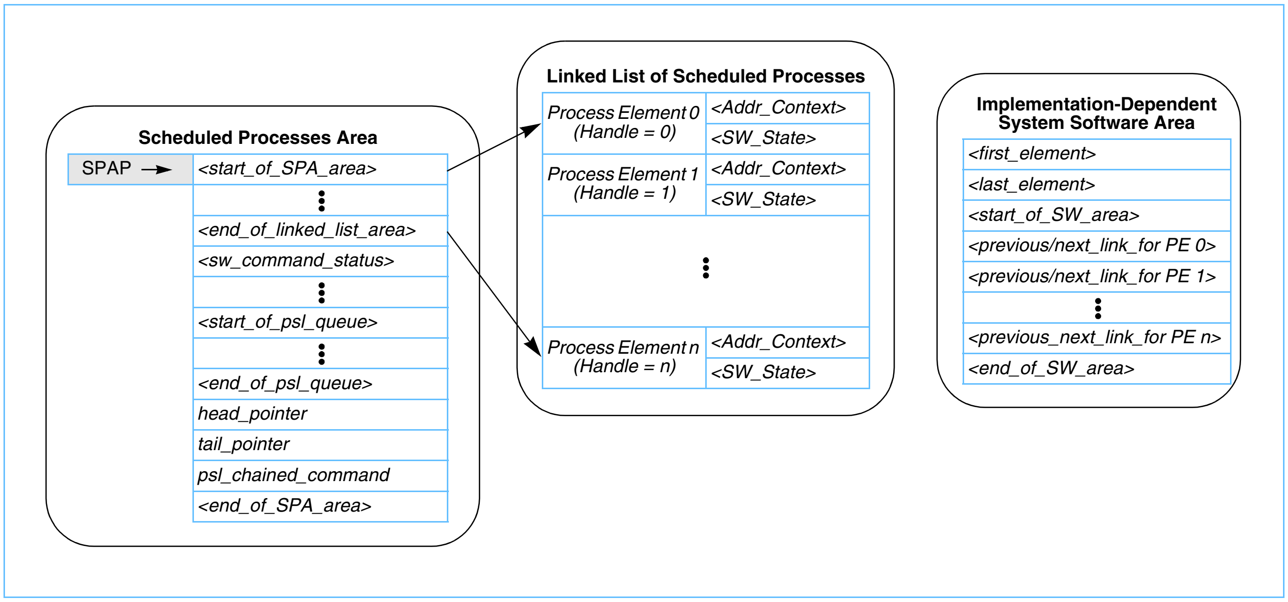


Table 2-1 defines the various fields and areas within the scheduled processes structure. The starting address of the area (SPA_Base) is defined by the PSL Scheduled Processes Area Pointer Register (PSL_SPAP_An). The size of the area (PSL_SPAP_An[size]) determines the number of process elements supported by the structure and the amount of storage that must be allocated. The storage must be contiguous in the real address space and naturally aligned to the size of the scheduled processes area.

Note: The structure for the scheduled processes in *Figure 2-3* contains an implementation-dependent system software area. This area is used to maintain the linked list pointers for maintaining the list of active process elements and the free list of process elements. How these pointers are maintained is implementation specific and outside the scope of the CAIA.

Table 2-1. Structure of Scheduled Processes Area (Page 1 of 2)

Mnemonic	Address (Byte)	Description
start_of_SPA_area	SPA_Base	This is the start of the area in system storage used by system software to store the linked list of process elements scheduled for the acceleration function units (AFUs). **Note:** The process elements in this area must never be cached by the PSL in a modified state.

Table 2-1. Structure of Scheduled Processes Area (Page 2 of 2)

Mnemonic	Address (Byte)	Description
end_of_linked_list_area	SPA_Base + (n × 128) - 1; where n = maximum number of process elements supported.	This is the end of the area in system storage used by system software to store the linked list of process elements scheduled for the AFUs.
Reserved	SPA_Base + ((N+1) x 128) SPA_Base + ((N+2) x 128); where n = maximum number of process elements supported.	Reserved area.
sw_command_status	SPA_Base + ((n+3) × 128); where n = maximum number of process elements supported.	Software command for the first PSL assigned to service the process element. The last PSL assigned to service the process elements returns the status. **Note:** This location must never be cached by the PSL in a modified state.
Note: Storage in the SPA above this address must <u>not</u> be read by system software.		
start_of_psl_queue_ area	SPA_Base + ((n+4) × 128); where n = maximum number of process elements supported.	This is the start of the area in system storage used by the PSLs for the queue of process elements waiting to run.
end_of_psl_queue_ area	SPA_Base + ((n+4) × 128) + (n × 8) - 1; where n = maximum number of process elements supported.	This is the end of the area in system storage used by the PSLs for the queue of process elements waiting to run.
head_pointer	SPA_Base + ((n+4) × 128) + ((((n × 8) + 127) >> 7) × 128); where n = maximum number of process elements supported.	Pointer to the next location to insert a preempted process element. The head pointer value is an index from the start address of the PSL queue area. **Note:** This location is aligned to the next cache line offset following the end of the PSL queue. If the number of cache lines needed for the PSL queue area is even, this location is the next cache line plus 1.
tail_pointer	SPA_Base + ((n+4) × 128) + ((((n × 8) + 127) >> 7) × 128) + 8; where n = maximum number of process elements supported.	Pointer to next process element to resume. The tail pointer value is an index from the start address of the PSL queue area.
psl_chained_command	SPA_Base + ((n+4) × 128) + ((((n × 8) + 127) >> 7) × 128) + 128; where n = maximum number of process elements supported.	Command for next PSL assigned to service the process elements.
end_of_SPA_area	SPA_Base + ((n+4) × 128) + ((((n × 8) + 127) >> 7) × 128) + 255; where n = maximum number of process elements supported.	End of the scheduled processes area.

2.3.1 Process Element Entry

Each process element entry is 128-bytes in length. *Table 2-2* shows the format of each process element. The shaded fields in *Table 2-2* correspond to privileged 1 registers, and the fields not shaded correspond to privileged 2 registers. The Software State field is an exception and does not have corresponding privileged 1 or privileged 2 registers.

Table 2-2. Process Element Entry Format

<table>
<tr><th rowspan="2">Word</th><th colspan="32">Process Element Entry</th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td colspan="32">State Register (0:31)</td></tr>
<tr><td>1</td><td colspan="32">State Register (32:63)</td></tr>
<tr><td>2</td><td>E</td><td>P</td><td colspan="30">SPOffset (most significant bits)</td></tr>
<tr><td>3</td><td colspan="20">SPOffset (least significant bits)</td><td colspan="6">Reserved</td><td colspan="6">SPSIZE</td></tr>
<tr><td>4</td><td colspan="4">R</td><td colspan="28">HTABORG (most significant bits)</td></tr>
<tr><td>5</td><td colspan="14">HTABORG (least significant bits)</td><td colspan="13">Reserved</td><td colspan="5">HTABSIZE</td></tr>
<tr><td>6</td><td colspan="4">R</td><td colspan="28">HAURP Physical Address (most significant bits)</td></tr>
<tr><td>7</td><td colspan="25">HAURP Physical Address (least significant bits)</td><td colspan="6">Reserved</td><td>V</td></tr>
<tr><td>8</td><td colspan="8">Reserved</td><td colspan="8">Idle_Time</td><td colspan="8">Reserved</td><td colspan="8">Context_Time</td></tr>
<tr><td>9</td><td colspan="16">IVTE_Offset_0</td><td colspan="16">IVTE_Offset_1</td></tr>
<tr><td>10</td><td colspan="16">IVTE_Offset_2</td><td colspan="16">IVTE_Offset_3</td></tr>
<tr><td>11</td><td colspan="16">IVTE_Range_0</td><td colspan="16">IVTE_Range_1</td></tr>
<tr><td>12</td><td colspan="16">IVTE_Range_2</td><td colspan="16">IVTE_Range_3</td></tr>
<tr><td>13</td><td colspan="32">LPID</td></tr>
<tr><td>14</td><td colspan="32">TID</td></tr>
<tr><td>15</td><td colspan="32">PID</td></tr>
<tr><td>16</td><td colspan="32">CSRP Effective Address (most significant bits)</td></tr>
<tr><td>17</td><td colspan="20">CSRP Effective Address (least significant bits)</td><td colspan="12">Limit</td></tr>
<tr><td>18</td><td colspan="2">B</td><td>Ks</td><td>Kp</td><td>N</td><td>L</td><td>C</td><td>Rsv</td><td colspan="2">LP</td><td colspan="22">Reserved</td></tr>
<tr><td>19</td><td colspan="18">Reserved</td><td colspan="14">AURP Virtual Address (most significant bits)</td></tr>
<tr><td>20</td><td colspan="32">AURP Virtual Address</td></tr>
<tr><td>21</td><td colspan="25">AURP Virtual Address (least significant bits)</td><td colspan="6">Reserved</td><td>V</td></tr>
<tr><td>22</td><td colspan="2">B</td><td>Ks</td><td>Kp</td><td>N</td><td>L</td><td>C</td><td>Rsv</td><td colspan="2">LP</td><td colspan="10">Reserved</td><td colspan="12">SegTableSize</td></tr>
<tr><td>23</td><td colspan="18">Reserved</td><td colspan="14">SSTP Virtual Address (most significant bits)</td></tr>
<tr><td>24</td><td colspan="32">SSTP Virtual Address</td></tr>
<tr><td>25</td><td colspan="24">SSTP Virtual address (least significant bits)</td><td colspan="7">Reserved</td><td>V</td></tr>
<tr><td>26</td><td colspan="32">Authority Mask (most significant bits)</td></tr>
<tr><td>27</td><td colspan="32">Authority Mask (least significant bits)</td></tr>
<tr><td>28</td><td colspan="32">Reserved</td></tr>
<tr><td>29</td><td colspan="32">Work Element Descriptor (WED word 0)</td></tr>
<tr><td>30</td><td colspan="32">Work Element Descriptor (WED word 1)</td></tr>
<tr><td>31</td><td colspan="32">Software State</td></tr>
</table>

2.3.2 Software State Field Format

The software state field in the process element is used by system software to indicate how the PSL should handle the process element.

This word in the process element must only be modified by system software.

V	Reserved	S	T
0	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31

Bits	Field Name	Description
0	V	Process element valid. 0 Process element information is not valid. 1 Process element information is valid.
1:29	Reserved	Reserved.
30	S	Suspend process element. 0 Process element can execute (not suspended). 1 Process element execution is suspended (suspended). All outstanding operations are also complete. **Note:** The process element can be added to the PSL queue even if the suspend flag is ‘1’.
31	T	Terminate process element. 0 Termination of the process element has not been requested. 1 Process element is being terminated.

2.3.3 Software Command/Status Field Format

There are two command/status words in the scheduled processes area; the sw_command_status word and the psl_chained_command word. These commands are used by system software and the PSLs to either terminate or to safely remove a process element.

Updates of the sw_command_status word by the PSL must be performed using a caching-inhibited write operation. In some implementations, a special write operation must be used. The special write operation allows the system to continue normal operation in the scenario where the CAIA-compliant device abnormally terminates while in the middle of the update operation.

Access Type	sw_command_status:	Read/write by both system software and PSL. **Note:** The PSL must never cache the line containing the sw_command_status word in a modified state.
	psl_chained_command:	Read/write by only the PSL.
Base Address Offset	sw_command_status:	SPA_Base + ((n + 3) × 128); where n = maximum number of process elements supported.
	psl_chained_command:	SPA_Base + ((n+4) × 128) + ((((n × 8) + 127) >> 7) × 128) + 128; where n = maximum number of process elements supported.

Command	Status
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

PSL_ID	Link
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:15	Command	Command. x‘0000’ No command. x‘0001’ terminate_element: Terminate the process element at the link provided. x‘0002’ remove_element: Remove the process element at the link provided. x‘0003’ suspend_element: Stop executing the process element at the link provided. x‘0004’ resume_element: Resume executing the process element at the link provided. x‘0005’ add_element: Software is adding a process element at the link provided. x‘0006’ update_element: Software is updating the process element state at the link provided. All other values are reserved. **Note:** The most significant bit of the command is reserved and must always be set to ‘0’.

Bits	Field Name	Description
16:31	Status	Status. The status field in the sw_command_status word must always be set to x‘0000’ by system software. The PSL should only update this field when setting the completion status. If the most significant bit is set, it indicates an error. For example, a status of 0x8001 indicates an error terminated a process element. x‘0000’ Operation pending. x‘0001’ Process element terminated. x‘0002’ Safe to remove process element from the linked list. x‘0003’ Process element suspended and all outstanding operations are complete. x‘0004’ Execution of the process element resumed. x‘0005’ PSL acknowledgment of an added process element. x‘0006’ PSL acknowledgment of an updated process element. ‘1ccc cccc cccc cccc’ Indicates an error with the requested command indicated by the “c” field. All other values are reserved.
32:47	PSL_ID	PSL identifier. The PSL identifier is used to select which PSL assigned to service the scheduled processes must perform the operation. When the sw_command_status word is written by system software, the PSL_ID must be the first in the list of PSLs assigned to service the processes. Each PSL has the ID of the next PSL in the list and forwards the command to the next PSL in the psl_chained_command if required.
48:63	Link	Process element link. The process element link is the offset from the SPA_Base, shifted right by 7 bits, of the process element to operate on.

2.4 Process Management

In the shared programming model, the PSL switches between the processes scheduled to use the AFUs by system software. This section describes the procedures for both system software and the PSLs for scheduling, descheduling, and terminating processes.

The CAIA allows for multiple PSLs or multiple “Virtual” PSLs to be assigned to a single Scheduled Processes Area containing a linked list of processes. When multiple PSLs are assigned to a linked list of processes, each PSL is given a unique ID as well as the ID of the next PSL, creating a chain of PSLs. When system software issues a process management command such as add_element, the PSL_ID field in the software command/status location is set to the ID of the first PSL in the chain and then issues an MMIO to th PSL_LLCMD_An Register to initiate the operation. If required, the PSL then sets the PSL_ID field in the the software command/status location to the ID of the next PSL in the chain, causing the next PSL to also perform the operation. In the following sections, the term first_psl_id refers to the ID of the first PSL in the chain and the term next_psl_id refers to the ID of the next PSL in the chain.

To schedule a process for an AFU, system software adds a process element entry to the linked list in system memory. Once added, the PSL starts the new process at the next available context interval for a time-sliced programming model or at an implementation-dependent point in time for an AFU-directed programming model.

For the time-sliced programming models, any newly added processes are placed into a circular queue maintained by the PSL, referred to as the psl_queue. Process elements are pulled from the psl_queue in a round-robin order by one or more CAIA-compliant devices to be run.

When a process element completes, system software is responsible for removing the process element and updating the link list before allocating the process element to another process.

To terminate a process element, system software first sets the system software state field in the process element to indicate that the process element is being terminated. Next, system software issues a termination command to the first PSL, which initiates a sequence of operations to remove the process element from the PSL queue. The termination pending status is needed to prevent a PSL from starting or resuming the process while the corresponding process entry is being removed from the PSL queue.

The following sections define the system software and PSL procedures for various process elements and linked list management. In the following sections, the terms starting with "link_of_element_to" refer to the process element link for which the command is to be performed. The process element link is the offset from the SPA_Base shifted right by seven bits.

- *Section 2.4.1 Adding a Process Element to the Linked List by System Software* on page 37
- *Section 2.4.2 PSL Queue Processing (Starting and Resuming Process Elements)* on page 40
- *Section 2.4.3 Terminating a Process Element* on page 41
- *Section 2.4.4 Removing a Process Element from the Linked List* on page 45
- *Section 2.4.5 Suspending a Process Element in the Linked List* on page 48
- *Section 2.4.6 Resuming a Process Element* on page 52
- *Section 2.4.7 Updating a Process Element in the Linked List* on page 54

2.4.1 Adding a Process Element to the Linked List by System Software

System software adds a new process element for each process that has work for the accelerator. The process element is added to the software-managed linked list of scheduled processes using the following sequence. The sequence outlined in *Section 2.4.1.1* is only for a single system-software process managing the linked list. Additional locking and synchronization steps are necessary to allow multiple system-software processes to concurrently manage the linked list.

2.4.1.1 Software Procedure

1. Determine if there is room in the linked list for the new process element.

   Note: The method that system software uses to calculate the free space in the linked list is implementation specific.
2. Write the new process state to a free process element location in the linked list area. The free process element can be obtained from a linked list of free processes or by some other implementation-specific means.
3. Set the valid flag in the software state to '1' (Software_State[V] = '1').
   Store x'80000000' to the 31st word of the process element to add.
4. Ensure that the terminate status is visible to all processes.
   System software running on the host processor must perform a **sync** instruction.
5. Write an add_element command to the software command/status field in the linked list area.
   Store (x'00050000' || first_psl_id || link_of_element_to_add) to address sw_command_status.
6. Update the system-software implementation-dependent free list and the process-element linked list structures to reflect the added process element.
7. Ensure that the new process element is visible to all processes.
   System software running on the host processor must perform a **sync** instruction.

8. Issue the add_element MMIO command to the first PSL.
   System software performs an MMIO to the PSL Linked List Command Register with the add_element command and the link to the new process being added.
   (PSL_LLCMD_An = x'000500000000' || link_of_element_to_add).
9. Wait for the PSLs to acknowledge the process element.
   - The process element is added when a load from sw_command_status returns (x'00050005' || first_psl_id || link_of_element_to_add).
   - If a value of all 1's is returned for the status, an error has occurred. An implementation-dependent recovery procedure should be initiated by hardware.

2.4.1.2 PSL Procedure for the Time-Sliced Programming Models

Note: Each PSL assigned to service the scheduled processes is configured with a unique identifier and the identifier of the next PSL in the list of PSLs servicing the processes. In addition, each PSL is identified as either the first PSL, the last PSL, both first and last PSL (only one PSL servicing the queue), or neither first or last PSL. The PSL ID Register contains the PSL unique identifier and the settings for first and last.

Operations Performed by the First PSL (PSL_ID[F] = '1')

When the add_element MMIO command is received by the first PSL, the PSL performs any operations necessary and sends the add_element command to the next PSL. The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

1. Performs a read of the cache line containing the head_pointer and tail_pointer, such that the cache line is owned by the PSL.
   - The PSL should prevent any other PSL from accessing the cache line until substep 3.
2. Writes the link to the process element and its status to the PSL queue of processes waiting to be restarted.
   - Writes the added process element link to the memory location pointed to by the head_pointer.
   - Adds 8 to the head_pointer; head_pointer = head_pointer + 8.
   - If the head_pointer is greater than end_of_PSL_queue_area,
     head_pointer = start_of_PSL_queue_area.
3. Releases the protection of the cache line containing the head_pointer and tail_pointer values.
4. The PSL sets the completion status in the software command/status field to indicate the process has been successfully added.
   - The status field in sw_command_status is set to x'0005' using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x'00050005' || first_psl_id || link_of_element_to_add).

2.4.1.3 PSL Procedure for the AFU-Directed Programming Models

Each PSL assigned to service the scheduled processes is configured with a unique identifier and the identifier of the next PSL in the list of PSLs servicing the processes. In addition, each PSL is identified as either the first PSL, the last PSL, both first and last PSL (only one PSL servicing the queue), or neither first or last PSL. The PSL ID Register contains the PSL unique identifier and the settings for first and last.

Operations Performed by the First PSL (PSL_ID[L,F] = ‘01’)

When the add_element MMIO command is received by the first PSL, the PSL performs any operations necessary and sends the add_element command to the next PSL. The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

1. The PSL notifies the AFU of the added process element. The AFU performs any necessary operations to prepare for the new process and then acknowledges the new process element. When the acknowledgment is received, the PSL continues with the next substep.
2. The PSL writes an add_element command to the psl_chained_command doubleword for the next PSL and watches for the add_element to be complete.
   - Write the value (x‘00050000’ || next_psl_id || link_of_element_to_add) to the psl_chained_command.

Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)

When the add_element command is detected by the next PSL, perform any operations necessary and send the add_element command to the next PSL. The add_element command is detected by monitoring the psl_chained_command doubleword. The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

1. The PSL notifies the AFU of the added process element. The AFU performs any necessary operations to prepare for the new process and then acknowledges the new process element. When the acknowledgment is received, the PSL continues with the next substep.
2. The next PSL writes an add_element command to the psl_chained_command doubleword for the next PSL and watches for the add_element to be complete.
   - Write the value (x‘00050000’ || next_psl_id || link_of_element_to_add) to the psl_chained_command.

Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)

When the *add_element* MMIO command is received or the *add_element* command is detected by the last PSL, perform any operations necessary and set the completion status in the software command/status word. The *add_element* command is detected by monitoring the *psl_chained_command* doubleword. The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

1. The PSL notifies the AFU of the added process element. The AFU performs any necessary operations to prepare for the new process and then acknowledges the new process element. When the acknowledgment is received, the PSL continues with the next substep.
2. The PSL sets the completion status in the software command/status field to indicate the process has been successfully added.

- The status field in sw_command_status is set to x‘0005’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00050005’ || first_psl_id || link_of_element_to_add).

2.4.2 PSL Queue Processing (Starting and Resuming Process Elements)

Multiple PSLs can be assigned to service the list of scheduled processes. Each PSL follows the sequence outlined in *Section 2.4.2.1* to start a new process or continue a previously started process. The following procedures apply only to the time-sliced programming models.

2.4.2.1 PSL Procedure for Time-Sliced Programming Models

1. **Check the PSL queue for processes waiting to be restarted.**
   a. Perform a read of the cache line containing the head_pointer and tail_pointer such that the cache line is owned by the PSL.
      - The PSL should prevent any other PSL from accessing the cache line until substep 3c or substep 3b.

   b. Compare the head and tail pointers of the PSL queue.
      - If the head_pointer does not equal the tail_pointer; a process is waiting to be started or resumed. Continue with step 3.
      - If the head_pointer equals the tail_pointer; no processes are waiting to be restarted. Continue with substep 1c.

   c. Release the protection of the cache line containing the head_pointer and tail_pointer values. Continue with step 2.
2. **No processes to run. Wait until a process is added to the PSL queue.**
   a. Wait for the head and tail pointers to be updated.
      - The PSL can either poll the cache line that contains the head_pointer and tail_pointer information for a change in state, or detect when the cache line is modified by another device using the coherency protocol.

   b. Continue with step 1.
3. **Start the next process in the PSL queue.**
   a. Remove the process to start/resume from the PSL queue.
      - Set the process element handle (PSL_PEHandle_An[PE_Handle]) for the process element to resume or start, and the process_state with the data contained at the tail_pointer.
      - Add 8 to the tail_pointer; tail_pointer = tail_pointer + 8.
      - If the tail_pointer is greater than end_of_PSL_queue_area, tail_pointer = start_of_PSL_queue_area.

   b. Release the protection of the cache line containing the head_pointer and tail_pointer values.

   c. Read the process element state from the linked list and start the process.
      - The process to start or resume is the value of the tail_pointer read in substep a.
      - If the suspend flag is set in the software status field, continue with substep 4.
      - If the suspend flag is not set in the software status field, perform a context restore if indicated by the process_state read in substep 3a and start the process. Continue with the next substep.

   d. Continue running the process until either the context time has expired or the process is completed.
      - If the processes are completed, continue with step 1.
      - If the context timer expires, request the AFU to perform a context save operation.

- If a context save is performed by the AFU, wait until the operation is completed and set process_state to indicate that a context restore is required. Continue with the step 4.

4. **Place the process element into the PSL queue of processes waiting to be started or resumed.**
   a. Perform a read of the cache line containing the head_pointer and tail_pointer such that the cache line is owned by the PSL.
      - The PSL should prevent any other PSL from accessing the cache line until substep c.

   b. Write the link to the process element and its status to the PSL queue of processes waiting to be restarted.
      - Write the process element handle (PSL_PEHandle_An[PE_Handle]) and the process_state to the memory location pointed to by the head_pointer.
      - Add 8 to the head_pointer; head_pointer = head_pointer + 8.
      - If the head_pointer is greater than end_of_PSL_queue_area, head_pointer = start_of_PSL_queue_area.

   c. Release the protection of the cache line that contains the head_pointer and tail_pointer values.

2.4.2.2 PSL Procedure for AFU-Directed Programming Models

The procedure for starting and resuming a process element in the AFU-directed programming models is implementation specific. In these models, system software adds a process element to the linked list and provides the application with a context handle. The lower 16-bits of the process handle are a pointer to the process element that contain the corresponding process state for the application. The AFU provides the lower 16-bits of the process handle (context ID) for each transaction associated with the process handle. The PSL uses the context ID to find the corresponding process element.

In the AFU-directed programming models, the PSL does not manage any queue of processes waiting to be resumed.

2.4.3 Terminating a Process Element

Under certain circumstances, system software might have to terminate a process element currently scheduled for the AFUs. Because a scheduled process element might have already been started or is currently being executed by a PSL, system software must follow the following sequence to safely terminate a process element in the linked list of scheduled processes.

2.4.3.1 Software Procedure

The following sequence is only for a single system software process managing the linked list. Additional locking and synchronization steps are necessary to allow for multiple system software processes to concurrently manage the linked list.

1. Set the terminate flag in the software state to ‘1’ (Software_State[T] = ‘1’).
   - Store x‘80000001’ to the 31st word of the process element to terminate.

2. Ensure that the *terminate* status is visible to all processes.
   - System software running on the host processor must perform a **sync** instruction.

3. Write a terminate_element command to the software command/status field in the linked list area.
   - Store (x‘00010000’ || first_psl_id || link_of_element_to_terminate) to address sw_command_status.

4. Ensure that the terminate_element command is visible to all processes.
   - System software running on the host processor must perform a **sync** instruction.

5. Issue the terminate_element MMIO command to the first PSL.
   - System software performs an MMIO to the PSL Linked List Command Register with the terminate_element command and the link of the process being terminated. (PSL_LLCMD_An = x'000100000000' || link_of_element_to_terminate).
6. Wait for the PSLs to complete the termination of the process element.
   - The process element is terminated when a load from sw_command_status returns (x'00010001' || first_psl_id || link_of_element_to_terminate).
   - If a value of all 1's is returned for the status, an error has occurred. An implementation-dependent recovery procedure should be initiated by hardware.
7. Reset the valid flag in the software state to '0' (Software_State[V] = '0').
   - Store x'00000000' to the 31st word of the process element to terminate.
8. Remove the process element from the linked list.
   - See the procedure in *Section 2.4.4 Removing a Process Element from the Linked List* on page 45.

2.4.3.2 PSL Procedure for Time-Sliced Programming Models

Each PSL assigned to service the scheduled processes is configured with a unique identifier and the identifier of the next PSL in the list of PSLs servicing the processes. In addition, each PSL is identified as either the first PSL, the last PSL, both first and last PSL (only one PSL servicing the queue), or neither first or last PSL. The PSL ID Register contains the PSL unique identifier and the settings for first and last.

Operations Performed by the First PSL (PSL_ID[L,F] = '01')

When the terminate_element MMIO command is received by the first PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sends the terminate_element command to the next PSL or sets the completion status in the software command/status word.

1. If the process element is running, the process is terminated. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully terminated. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
   - The status field in sw_command_status is set to x'0001' using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x'00010001' || first_psl_id || link_of_element_to_terminate).
2. If the process element is not running, the PSL writes a termination command to the psl_chained_command doubleword for the next PSL and watches for the termination to be complete.
   - Write the value (x'00010000' || next_psl_id || link_of_element_to_terminate) to the psl_chained_command doubleword.
   - While waiting for the process to be terminated, the PSL does not attempt to start the corresponding process or any process with the complete, suspend, or terminate flags set. The PSL can perform other operations.
     - The process is terminated when the status field in sw_command_status is x'0001'.

Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)

When the terminate_element command is detected by the next PSL, the PSL checks to see if the process element that is being terminated is currently running. It performs any operations necessary, and sends the terminate_element command to the next PSL or sets the completion status in the software command/status word. The terminate_element command is detected by monitoring the psl_chained_command doubleword.

1. If the process element is running, the process is terminated. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully terminated. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
   - The status field in sw_command_status is set to x‘0001’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00010001’ || first_psl_id || link_of_element_to_terminate).
2. If the process element is not running, the PSL writes a termination command to the psl_chained_command doubleword for the next PSL and watches for the termination to be completed.
   - Write the value (x‘00010000’ || next_psl_id || link_of_element_to_terminate) to psl_chained_command.
   - While waiting for the process to be terminated, the PSL does not attempt to start the corresponding process or any process with the complete, suspend, or terminate flags set. The PSL can perform other operations.
     – The process is terminated when the status field in sw_command_status is x‘0001’.

Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)

When the terminate_element MMIO command is received or the terminate_element command is detected by the last PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sets the completion status in the software command/status word. The terminate_element command is detected by monitoring the psl_chained_command doubleword.

1. If the process element is running, the process is terminated and the PSL sets the completion status in the software command/status field to indicate that the process has been successfully terminated. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
   - The status field in sw_command_status is set to x‘0001’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00010001’ || first_psl_id || link_of_element_to_terminate).
2. If the process element is not running, the PSL searches the queue to determine if the process is waiting to be resumed and indicates that the process termination is complete.
   - The PSL pulls each process link from the PSL queue and compares the link with the process being terminated. The full queue is searched.
     (1) Performs a read of the cache line containing the head_pointer and tail_pointer such that the cache line is owned by the PSL.
       – The PSL should prevent any other PSL from accessing the cache line until substep 6.
         - Save the head_pointer location to an initial_head_pointer internal register.
     (2) Removes the process from the PSL queue.
       – Read the process_element_link and process_state pointed to by the tail_pointer.

- Add 8 to the tail_pointer; tail_pointer = tail_pointer + 8.
- If the tail_pointer is greater than end_of_PSL_queue_area, tail_pointer = start_of_PSL_queue_area.

(3) Compares the process_element_link read in substep 2 with link_of_element_to_terminate.

- If the links match, continue with substep 5.
- If the link do not match, continue with the next substep.

(4) Puts the process_element_link and process_state back on the PSL queue.

- Writes the process_element_link and process_state to the memory location pointed to by the head_pointer.
- Add 8 to the head_pointer; head_pointer = head_pointer + 8.
- If the head_pointer is greater than end_of_PSL_queue_area, head_pointer = start_of_PSL_queue_area.

(5) Compares the tail_pointer to the initial_head_pointer.

- If the tail_pointer is not equal to initial_head_pointer, continue with substep 2.
- If the tail_pointer is equal to initial_head_pointer, continue with the next substep.

(6) Release the protection of the cache line containing the head_pointer and tail_pointer values.

- After completing the search of all process links, the status field in sw_command_status is set to x‘0001’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00010001’ || first_psl_id || link_of_element_to_terminate).

  All other PSLs can now stop protecting against starting the process being terminated.

2.4.3.3 PSL Procedure for AFU-Directed Programming Models

Each PSL assigned to service the scheduled processes is configured with a unique identifier and the identifier of the next PSL in the list of PSLs servicing the processes. In addition, each PSL is identified as either the first PSL, the last PSL, both first and last PSL (only one PSL servicing the queue), or neither first or last PSL. The PSL ID Register contains the PSL unique identifier and the settings for first and last.

Operations Performed by the First PSL (PSL_ID[L,F] = ‘01’)

When the terminate_element MMIO command is received by the first PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sends the *terminate_element* command to the next PSL.

1. The PSL notifies the AFU of the process element termination. The AFU performs any necessary operations to remove the process and then acknowledges the termination of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. If the process is running, the process is terminated. The AFU and PSL are allowed to complete any outstanding transactions but should not start any new transactions for the process.
3. The PSL writes a termination command to the psl_chained_command doubleword for the next PSL.
   - Write the value (x‘00010000’ || next_psl_id || link_of_element_to_terminate) to the psl_chained_command.

Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)

When the terminate_element command is detected by the next PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sends the terminate_element command to the next PSL. The terminate_element command is detected by monitoring the psl_chained_command doubleword.

1. The PSL notifies the AFU of the process element termination. The AFU performs any necessary operations to remove the process and then acknowledges the termination of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. If the process is running, the process is terminated. The AFU and PSL are allowed to complete any outstanding transactions but should not start any new transactions for the process.
3. The PSL writes a termination command to the psl_chained_command doubleword for the next PSL and watches for the termination to be complete.
   - Write the value (x‘00010000’ || next_psl_id || link_of_element_to_terminate) to the *psl_chained_command*.

Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)

When the terminate_element MMIO command is received or the terminate_element command is detected by the last PSL, the PSL checks to see if the process element that is being terminated is currently running. It performs any operations necessary, and sets the completion status in the software command/status word. The terminate_element command is detected by monitoring the psl_chained_command doubleword.

1. The PSL notifies the AFU of the process element termination. The AFU performs any necessary operations to remove the process and then acknowledges the termination of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. If the process is running, the process is terminated. The AFU and PSL are allowed to complete any outstanding transactions but should not start any new transactions for the process.
3. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully terminated.
   - The status field in sw_command_status is set to x‘0001’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00010001’ || first_psl_id || link_of_element_to_terminate).

2.4.4 Removing a Process Element from the Linked List

To make room for new process elements in the linked list, completed and terminated process elements must be removed by system software. To safely remove a process element from the linked list of scheduled processes, software must follow the sequence outlined in *Section 2.4.4.1*.

Implementation Note:

The removal of a process element must also invalidate all cache copies of translations that are associated with the process element being removed. An implementation cannot depend on system software performing TLB and SLB invalidates.

2.4.4.1 Software Procedure

Note: The following sequence is only for a single system-software process managing the linked list. Additional locking and synchronization steps are necessary to allow for multiple system-software processes to concurrently manage the linked list.

1. Update the system-software implementation-dependent free list and process-element linked list structures to reflect the removal of the process element.
2. Write a remove_element command to the software command/status field in the linked list area.
   - Store (x‘00020000’ || first_psl_id || link_of_element_to_remove) to sw_command_status.
3. Ensure that the remove_element command is visible to all processes.
   - System software running on the host processor must perform a **sync** instruction.
4. Issue the remove_element MMIO command to the first PSL.
   - System software performs an MMIO to the PSL Linked List Command Register with the remove_element command and the link of the process being removed. (PSL_LLCMD_An = x‘000200000000’ || link_of_element_to_remove)
5. Wait for the PSLs to acknowledge the removal of the process element.
   - The process element is terminated when a load from sw_command_status returns (x‘00020002’ || first_psl_id || link_of_element_to_remove).
   - If a value of all 1’s is returned for the status, an error has occurred. An implementation-dependent recovery procedure should be initiated by hardware.
6. Invalidate the PSL SLBs and TLBs for the processes being removed.
   - System software performs an MMIO write to the Lookaside Buffer Invalidation Selector with the process ID and logical partition ID of the process being removed. (PSL_LBISEL = PID || LPID).
   - System software performs an MMIO write to invalidate the SLBs (PSL_SLBIA = x‘3’).
   - System software waits until the SLB invalidate is completed (MMIO read of PSL_SLBIA returns zero in the least significant bit).
   - System software performs an MMIO write to invalidate the TLBs (PSL_TLBIA = x‘3’).
   - System software waits until the TLB invalidate is completed (MMIO read of PSL_TLBIA returns zero in the least significant bit).
7. At this point, the memory locations for the process element that was removed can now be reused.

2.4.4.2 PSL Procedure for Time-Sliced Programming Models

Operations Performed by the First PSL (PSL_ID[F] = ‘1’)

When the *r*emove_element MMIO command is received by the first PSL, the PSL sets the completion status in the software command/status word.

1. The PSL sets the completion status in the software command/status field to indicate that it is now safe to remove the process element from the linked list.
   - The status field in sw_command_status is set to x‘0002’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00020002’ || first_psl_id || link_of_element_to_remove).

2.4.4.3 PSL Procedure for AFU-Directed Programming Models

Operations Performed by the First PSL (PSL_ID[L,F] = ‘01’)

When the remove_element MMIO command is received by the first PSL, the PSL notifies the AFU that the process element is being removed and sends the remove_element command to the next PSL.

1. The PSL notifies the AFU of the process element removal. The AFU performs any necessary operations to remove the process and then acknowledges the removal of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. The PSL sets the completion status in the software command/status field to indicate that it is now safe to remove the process element from the linked list.
   - Write the value (x‘00020000’ || next_psl_id || link_of_element_to_remove) to the psl_chained_command.
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)

When the remove_element command is detected by the next PSL, the PSL notifies the AFU that the process element is being removed and sends the remove_element command to the next PSL. The remove_element command is detected by monitoring the psl_chained_command doubleword.

1. The PSL notifies the AFU of the process element removal. The AFU performs any necessary operations to remove the process and then acknowledges the removal of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. The PSL sets the completion status in the software command/status field to indicate that it is now safe to remove the process element from the linked list.
   - Write the value (x‘00020000’ || next_psl_id || link_of_element_to_remove) to the psl_chained_command.
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)

When the suspend_element MMIO command is received or the suspend_element command is detected by the last PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sets the completion status in the software command/status word. The suspend_element command is detected by monitoring the psl_chained_command doubleword.

1. The PSL notifies the AFU of the process element removal. The AFU performs any necessary operations to remove the process and then acknowledges the removal of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully removed.
   - The status field in sw_command_status is set to x‘0002’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails.

2.4.5 Suspending a Process Element in the Linked List

The suspend flag in the software process element state is used to temporarily stall the processing of a process element. If the process is already running on an AFU, setting the suspend flag stops the currently running process.

2.4.5.1 Software Procedure

1. Set the suspend flag in software_state to ‘1’ (Software_State[S] = ‘1’).
   - Store x‘80000002’ to the 31st word of the process element to suspend.
2. Ensure that the update to software_state is visible to all processes.
   - System software running on the host processor must perform a **sync** instruction.
3. Write a suspend_element command to the software command/status field in the linked list area.
   - Store (x‘00030000’ || first_psl_id || link_of_element_to_suspend) to sw_command_status.
4. Ensure that the suspend_element command is visible to all processes.
   - System software running on the host processor must perform a **sync** instruction.
5. Issue the suspend_element MMIO command to the first PSL.
   - System software performs an MMIO to the PSL Linked List Command Register with the suspend_element command and the link to the new process being added. (PSL_LLCMD_An = x‘000300000000’ || link_of_element_to_suspend).
6. Wait for the PSL to suspend the process element.
   - The process element is suspended when a load from sw_command_status returns (x‘00030003’ || first_psl_id || link_of_element_to_suspend).
   - If a value of all 1’s is returned for the status, an error has occurred. An implementation-dependent recovery procedure should be initiated by hardware.

2.4.5.2 PSL Procedure for Time-Sliced Programming Models

Each PSL assigned to service the scheduled processes is configured with a unique identifier and the identifier of the next PSL in the list of PSLs servicing the processes. In addition, each PSL is identified as either the first PSL, the last PSL, both first and last PSL (only one PSL servicing the queue), or neither first or last PSL. The PSL ID Register contains the PSL unique identifier and the settings for first and last.

Operations Performed by the First PSL (PSL_ID[L,F] = ‘01’)

When the suspend_element MMIO command is received by the first PSL, the PSL checks to see if the process element being suspended is currently running. It performs any operations necessary, and sends the suspend_element command to the next PSL or sets the completion status in the software command/status word.

1. If the process is running, the process is suspended. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully suspended. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
   - The status field in sw_command_status is set to x‘0003’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00030003’ || first_psl_id || link_of_element_to_suspend).
2. If the process element is not running, the PSL writes a suspend command to the psl_chained_command doubleword for the next PSL.

- Write the value (x‘00030000’ || next_psl_id || link_of_element_to_suspend) to the psl_chained_command.
- The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)

When the suspend_element command is detected by the next PSL, the PSL checks to see if the process element being suspended is currently running. It performs any operations necessary, and sends the suspend_element command to the next PSL or sets the completion status in the software command/status word. The suspend_element command is detected by monitoring the psl_chained_command doubleword.

1. If the process element is running, the process is suspended. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully suspended. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
    - The status field in sw_command_status is set to x‘0003’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00030003’ || first_psl_id || link_of_element_to_suspend).
2. If the process element is not running, the PSL writes a suspend command to the psl_chained_command doubleword for the next PSL.
    - Write the value (x‘00030000’ || next_psl_id || link_of_element_to_suspend) to the psl_chained_command.
    - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)

When the suspend_element MMIO command is received or the suspend_element command is detected by the last PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sets the completion status in the software command/status word. The suspend_element command is detected by monitoring the psl_chained_command doubleword.

1. If the process element is running, the process is suspended. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
2. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully suspended.
    - The status field in sw_command_status is set to x‘0003’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00030003’ || first_psl_id || link_of_element_to_suspend).
    - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

2.4.5.3 PSL Procedure for AFU-Directed Programming Models

Each PSL assigned to service the scheduled processes is configured with a unique identifier and the identifier of the next PSL in the list of PSLs servicing the processes. In addition, each PSL is identified as either the first PSL, the last PSL, both first and last PSL (only one PSL servicing the queue), or neither first or last PSL. The PSL ID Register contains the PSL unique identifier and the settings for first and last.

Operations Performed by the First PSL (PSL_ID[L,F] = '01')

When the suspend_element MMIO command is received by the first PSL, the PSL checks to see if the process element being suspended is currently running. It performs any operations necessary, and sends the suspend_element command to the next PSL or sets the completion status in the software command/status word.

1. The PSL notifies the AFU of the suspended process element. The AFU performs any necessary operations to suspend the process and then acknowledges the suspension of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. If the process is running, the process is suspended. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully suspended. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
   - The status field in sw_command_status is set to x'0003' using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x'00030003' || first_psl_id || link_of_element_to_suspend).
3. If the process element is not running, the PSL writes a suspend command to the psl_chained_command doubleword for the next PSL.
   - Write the value (x'00030000' || next_psl_id || link_of_element_to_suspend) to the psl_chained_command.
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Operations Performed by the Next PSL (PSL_ID[L,F] = '00')

When the suspend_element command is detected by the next PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sends the terminate_element command to the next PSL or sets the completion status in the software command/status word. The suspend_element command is detected by monitoring the psl_chained_command doubleword.

1. The PSL notifies the AFU of the suspended process element. The AFU performs any necessary operations to suspend the process and then acknowledges the suspension of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. If the process element is running, the process is suspended. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
3. The PSL writes a suspend command to the psl_chained_command doubleword for the next PSL.
   - Write the value (x'00030000' || next_psl_id || link_of_element_to_suspend) to the psl_chained_command.
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)

When the suspend_element MMIO command is received or the suspend_element command is detected by the last PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sets the completion status in the software command/status word. The suspend_element command is detected by monitoring the psl_chained_command doubleword.

1. The PSL notifies the AFU of the suspended process element. The AFU performs any necessary operations to suspend the process and then acknowledges the suspension of the process element. When the acknowledgment is received, the PSL continues with the next substep.
2. If the process element is running, the process is suspended. The PSL is allowed to complete any outstanding transactions but should not start any new transactions for the process.
3. The PSL sets the completion status in the software command/status field to indicate the process has been successfully suspended.
   - The status field in sw_command_status is set to x‘0003’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00030003’ || first_psl_id || link_of_element_to_suspend).
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

2.4.6 Resuming a Process Element

The procedure for resuming a process element is used to restart the execution of a process element after the process has been suspended.

2.4.6.1 Software Procedure

1. Reset the suspend flag in the software state to ‘0’ (Software_State[S] = ‘0’).
   - Store x‘80000000’ to the 31st word of the process element to suspend.
2. Ensure that the update to the software_state is visible to all processes.
   - System software running on the host processor must perform a **sync** instruction.
3. Write the resume_element command to the software command/status field in the linked list area.
   - Store (x‘00040000’ || first_psl_id || link_of_element_to_resume) to sw_command_status.
4. Ensure that the resume_element command is visible to all processes.
   - System software running on the host processor must perform a **sync** instruction.
5. Issue the resume_element MMIO command to the first PSL.
   - System software performs an MMIO to the PSL Linked List Command Register with the update_element command and the link to the new process being added. (PSL_LLCMD_An = x‘000400000000’ || link_of_element_to_resume).
6. Wait for the PSLs to acknowledge the update of the process element.
   - The process element is updated when a load from sw_command_status returns (x‘00040004’ || first_psl_id || link_of_element_to_resume).
   - If a value of all 1’s is returned for the status, an error has occurred. An implementation-dependent recovery should be initiated by hardware.

2.4.6.2 PSL Procedure for Time-Sliced and AFU-Directed Programming Models

Each PSL assigned to service the scheduled processes is configured with a unique identifier and the identifier of the next PSL in the list of PSLs servicing the processes. In addition, each PSL is identified as either the first PSL, the last PSL, both first and last PSL (only one PSL servicing the queue), or neither first or last PSL. The PSL ID Register contains the PSL unique identifier and the settings for first and last.

Operations Performed by the First PSL (PSL_ID[L,F] = ‘01’)

When the *resume_element* MMIO command is received by the first PSL, the PSL performs any operations necessary and sends the *resume_element* command to the next PSL. The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

1. When operating in an AFU-directed programming model, the PSL notifies the AFU of the process element being resumed. The AFU performs any necessary operations to resume execution of the process and then acknowledges the resumed process element. When the acknowledgment is received, the PSL continues with the next substep. The AFU is not notified of the added process element for all other programming models.

2. The PSL writes an resume_element command to the psl_chained_command doubleword for the next PSL.
   - Write the value (x‘00040000’ || next_psl_id || link_of_element_to_resume) to the psl_chained_command.

Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)

When the resume_element command is detected by the next PSL, the PSL performs any operations necessary and sends the resume_element command to the next PSL. The resume_element command is detected by monitoring the psl_chained_command doubleword. The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

1. When operating in an AFU-directed programming model, the PSL notifies the AFU of the process element being resumed. The AFU performs any necessary operations to resume execution of the process and then acknowledges the resumed process element. When the acknowledgment is received, the PSL continues with the next substep. The AFU is not notified of the resumed process element for any other programming models.
2. The PSL writes an resume_element command to the psl_chained_command doubleword for the next PSL.
   - Write the value (x‘00040000’ || next_psl_id || link_of_element_to_resume) to the psl_chained_command.

Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)

When the resume_element MMIO command is received or the resume_element command is detected by the last PSL, the PSL performs any operations necessary and sets the completion status in the software command/status word. The resume_element command is detected by monitoring the psl_chained_command doubleword. The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

1. When operating in an AFU-directed programming model, the PSL notifies the AFU of the process element being resumed. The AFU performs any necessary operations to resume execution of the process and then acknowledges the resumed process element. When the acknowledgment is received, the PSL continues with the next substep. The AFU is not notified of the resumed process element for any other programming models.
2. The PSL sets the completion status in the software command/status field to indicate that the process has been successfully resumed.
   - The status field in sw_command_status is set to x‘0004’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00040004’ || first_psl_id || link_of_element_to_resume).

2.4.7 Updating a Process Element in the Linked List

The update flag in the software process element state is used to update the state of a process element. This command causes the PSL to invalidate any noncoherent copies of the process element that might be cached and to read a new copy from system memory. If the update of the process element is required to be atomic, the process element must be suspended before the update is made (suspend, update, resume).

2.4.7.1 Software Procedure

1. Write the *update_element* command to the software command/status field in the linked list area.
   - Store (x‘00060000’ || first_psl_id || link_of_element_to_update) to sw_command_status.
2. Ensure that the update_element command is visible to all processes.
   - System software running on the host processor must perform a **sync** instruction.
3. Issue the update_element MMIO command to the first PSL.
   - System software performs an MMIO to the PSL Linked List Command Register with the update_element command and the link to the new process being added. (PSL_LLCMD_An = x‘000600000000’ || link_of_element_to_update).
4. Wait for the PSL to acknowledge the update of the process element.
   - The process element is updated when a load from sw_command_status returns (x‘00060006’ || first_psl_id || link_of_element_to_update).
   - If a value of all 1’s is returned for the status, an error has occurred. An implementation-dependent recovery procedure should be initiated by hardware.

2.4.7.2 PSL Procedure for Time-Sliced and AFU-Directed Programming Models

Each PSL assigned to service the scheduled processes is configured with a unique identifier and the identifier of the next PSL in the list of PSLs servicing the processes. In addition, each PSL is identified as either the first PSL, the last PSL, both first and last PSL (only one PSL servicing the queue), or neither first or last PSL. The PSL ID Register contains the PSL unique identifier and the settings for first and last.

Operations Performed by the First PSL (PSL_ID[L,F] = ‘01’)

When the update_element MMIO command is received by the first PSL, the PSL checks to see if the process element being updated is currently running. It performs any operations necessary, and sends the update_element command to the next PSL.

1. When operating in an AFU-directed programming model, the PSL notifies the AFU of the updated process element. The AFU performs any necessary operations to update the process and then acknowledges the updated process element. When the acknowledgment is received, the PSL continues with the next substep. The AFU is not notified of the updated process element for any other programming models.
2. If the process is running, the PSL completes any outstanding transactions and does not start any new transactions for the process. The PSL then invalidates the process element state and refetches a new copy from the process element linked list in system memory. If the process element is coherently cached, the update is automatically handled by the coherency protocol.
   - The status field in sw_command_status is set to x‘0006’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00060006’ || first_psl_id || link_of_element_to_update).

- The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

3. If the process element is not running, the PSL writes an update command to the psl_chained_command doubleword for the next PSL.
   - Write the value (x‘00060000’ || next_psl_id || link_of_element_to_update) to the psl_chained_command.
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Operations Performed by the Next PSL (PSL_ID[L,F] = ‘00’)

When the update_element command is detected by the next PSL, the PSL checks to see if the process element being updated is currently running. It performs any operations necessary, and sends the terminate_element command to the next PSL or sets the completion status in the software command/status word. The update_element command is detected by monitoring the psl_chained_command doubleword.

1. When operating in an AFU-directed programming model, the PSL notifies the AFU of the updated process element. The AFU performs any necessary operations to update the process and then acknowledges the updated process element. When the acknowledgment is received, the PSL continues with the next substep. The AFU is not notified of the updated process element for any other programming models.
2. If the process is running, the PSL completes any outstanding transactions and does not start any new transactions for the process. The PSL then invalidates the process element state and refetches a new copy from the process element linked list in system memory. If the process element is coherently cached, the update is automatically handled by the coherency protocol.
   - The status field in sw_command_status is set to x‘0006’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00060006’ || first_psl_id || link_of_element_to_update).
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.
3. If the process element is not running, the PSL writes a suspend command to the psl_chained_command doubleword for the next PSL.
   - Write the value (x‘00060000’ || next_psl_id || link_of_element_to_terminate) to the psl_chained_command.
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Operations Performed by the Last PSL (PSL_ID[L] = ‘1’)

When the update_element MMIO command is received or the update_element command is detected by the last PSL, the PSL checks to see if the process element being terminated is currently running. It performs any operations necessary, and sets the completion status in the software command/status word. The suspend_element command is detected by monitoring the psl_chained_command doubleword.

1. When operating in an AFU-directed programming model, the PSL notifies the AFU of the updated process element. The AFU performs any necessary operations to update the process and then acknowledges the updated process element. When the acknowledgment is received, the PSL continues with the next substep. The AFU is not notified of the updated process element for any other programming models.

2. If the process is running, the PSL completes any outstanding transactions and does not start any new transactions for the process. The PSL then invalidates the process element state and refetches a new copy from the process element linked list in system memory. If the process element is coherently cached, the update is automatically handled by the coherency protocol.
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.
3. The PSL sets the completion status in the software command/status field to indicate the process has been successfully suspended.
   - The status field in sw_command_status is set to x‘0006’ using a caching-inhibited DMA or special memory-update operation that is guaranteed not to corrupt memory if the operation fails. The final value of sw_command_status should be (x‘00060006’ || first_psl_id || link_of_element_to_update).
   - The PSL does not start any process with a software state of complete, suspend, or terminate. A process element with the suspend flag set can be added to the PSL queue.

Part I. POWER Service Layer

Section 3 through *Section 8* describe the facilities that are provided by the Coherent Accelerator Interface Architecture (CAIA) for operating environment software. These facilities include an operating system or a hypervisor and the management of the accelerator function units (AFUs). They should only be made available to software running in a privileged mode, either privilege 1 mode or privilege 2 mode. Collectively, these facilities are called the POWER Service Layer (PSL).

3. Privileged Mode Overview

Each of the privileged facilities and their related registers is discussed in the following sections.

- *Section 4 Power ISA (Book III) Compatibility* on page 59
- *Section 7 Storage Addressing* on page 69
- *Section 8 PSL Privileged Facilities* on page 73

Table A-2. PSL Privilege 1 Memory Map on page 173, *Table A-3. PSL Slice Privilege 1 Memory Map* on page 175, and *Table A-4. PSL Slice Privilege 2 Memory Map* on page 176 list the privileged registers.

3.1 Privileged-Mode Organization

The facilities described in the privileged-mode environment are classified as either privilege 1 or privilege 2. These designations relate to a suggested hierarchy of privileged access. The access hierarchy is defined to support a 2-level operating environment. An example of such an operating environment is when multiple operating systems run concurrently on top of a more privileged hypervisor. This type of operating environment implements logical partitioning.

Privilege 1 registers are the most privileged. They are intended to be accessed by a hypervisor or by firmware operating in hypervisor mode[2] (HV = ‘1’ and PR = ‘0’) when supporting logical partitioning. Privilege 2 registers are intended for privileged operating-system code running in the HV = ‘0’ and PR = ‘0’ mode. When a single-level operating environment exists, firmware and the privileged operating system typically combine privilege 1 and privilege 2 resources into one privileged level.

The page table must be set up so that only privileged mode software can access the facilities in the privileged-mode environment. Problem-state software must never be allowed access to any facilities that are defined in the privileged-mode environment. The PSL does not check that an access to a privileged-mode facility is performed by a privileged-mode process. This authorization checking is provided by the host processor’s page protection mechanism.

2. For information about hypervisor mode, see the *Power ISA.*

4. Power ISA (Book III) Compatibility

This section covers the compatibility of the POWER Service Layer used in a CAIA-compliant accelerator with the privileged facilities defined in *Power ISA, Book III*.

4.1 Optional Features in Power ISA, Book III

There are no optional features in *Power ISA Architecture, Book III* that are required by the CAIA.

4.2 Incompatibilities with Power ISA, Book III

There are no incompatibilities with the *Power ISA Architecture, Book III* in the CAIA.

4.3 Extensions to the Power ISA, Book III

There are no extensions to the *Power ISA Architecture, Book III* that are required by the CAIA.

5. Context Management

In the shared and dedicated programming models, the POWER Service Layer (PSL) manages switching between the scheduled processes in the process element linked list. For these two programming models, each process is given a programmable amount of time to execute a job.

In the AFU-directed programming model, the AFU supplies a process handle to the PSL for selection of the process state (that is, the process element) to use for the main storage request.

5.1 Time-Sliced Context Management Procedure

The basic procedure performed by the PSL and AFU for processing the scheduled processes is as follows:

1. The PSL reads the next process element from the scheduled processes area. (See *Section 2.4.2 PSL Queue Processing (Starting and Resuming Process Elements)* on page 40)
2. The PSL updates the associated privileged 1 and privileged 2 registers.
3. The PSL requests that the AFU restore its state if required.
4. The AFU restores the state if required.
5. The PSL starts the AFU to perform the requested task.
6. If the AFU task completes or the AFU yields the rest of the context time, continue with step 8.
7. After a configurable time (PSL_CtxTime_An[Context_Time]), the PSL preempts the AFU.
8. When at the precise state, the AFU saves the state if required.
9. The PSL updates the utilization records (HAUR and AUR).
10. If the AFU's task is not complete, the PSL writes a link to the process element and the status to the PSL queue of processes waiting to be restarted.
11. The PSL resets the AFU and continues with step 1.

In step 7, the PSL sends a preemption request to the AFU. In response, the AFU attempts to reach a precise state where the job can be restarted. If needed, the AFU can request that the state be saved to system memory (step 8). After the AFU has reached the precise state and saved the context (if required), the AFU must indicate to the PSL that the AFU is idle and a new context can now be started (AFU_Idle).

After sending the preemption request in step 8, the PSL waits a predetermined time period (PSL_CtxTime[Warn_OS]) for the AFU_idle indication. If the AFU_idle indication is not received during this time period, an interrupt is sent to the operating system. The action taken by the operating system is operating-system implementation specific.

After the PSL_CtxTime[Warn_OS] time period expires, the PSL waits another predetermined time period (PSL_CtxTime[Warn_Hypervisor]) for the AFU_idle indication. If the AFU_idle indication is not received during this second time period, an interrupt is sent to the hypervisor. The action taken by the hypervisor is implementation specific but must include purging the current active job.

The AFU is given a predetermined amount of time (PSL_CtxTime_An[Idle_Time]) from the preemption in step 7 to indicate AFU_idle. After this time period expires, the PSL starts to track the amount of time (in PSL_CtxTime_An[Context_Time] units) until the AFU_idle indication is received. The number of units is saved as part of the status in the PSL queue of processes waiting to be restarted in step 10. This value is

then used to skip process elements that exceeded their allocation of the AFU. This is achieved by evaluating the queue entry in step 1. If the number of units in the status field is not zero, the PSL decrements the number of units by 1, and the procedure continues with step 1.

Note: For the PSL to reach step 9, all pending reads and writes of main storage must be complete. The PSL does not start a new process element until all pending reads and writes are complete. This includes any outstanding segment, page faults, or storage exceptions.

6. Interrupts

The PSL uses the PCIe message signaled interrupt (MSI-X) mechanism for the presentation of all interrupts. The MSI-X address and data are calculated by the PSL from values in the PSL_IVTE_Offset_An and PSL_IVTE_Limit_An Registers or the corresponding process element entries. There are three modes for calculating the address and data: fixed platform-specific address, single MSI-X table entry, and full MSI-X table. The mode supported by the PSL is provided in the status field (bit 14:13) of the vendor-specific extended configuration capability (VSEC) structure.

When operating with a fixed platform-specific address or a single MSI-X table entry, the PSL posts an interrupt using a PCIe posted write operation to address (MSI-X_Base_Addr | (x'00000000000' || IVTE || x'0') with data (x'0000000000000000'); where the IVTE value is calculated using the equation in *Section 8.1.15* on page 95. The MSI-X_Base_Address is either a fixed 64-bit platform-specific value or the 64-bit MSI-X address at entry 0 of the MSI-X table. By performing this operation, the IVTE value directly indexes into an interrupt vector table stored in system memory. The data for the MSI-X operation is always zero. In this mode, the pending bit array (PBA) is not used.

When operating with a full MSI-X table, MSI-X interrupts are presented to the system using a posted write with an address and data specified from an MSI-X table. For a CAIA-compliant device, the index into the MSI-X table to select the address and data is the IVTE. The IVTE is calculated using the equation in *Section 8.1.15* on page 95. In this mode, the pending bit array is also used.

Implementation Note:

In a POWER system, an MSI-X interrupt is presented using a 64-bit address of x'10000000000vvvv0'. The 16-bit value represented by 'v' is the interrupt vector table entry (IVTEn). For the single MSI-X entry mode, entry 0 provides a means of setting the base address.

The PSL can present a programmable number of interrupts (n) per context (process element) and one error interrupt. The logical interrupt source number (LISNn) presented to the PSL by the AFU is converted into a 16-bit interrupt vector table entry (IVTE) offset using the PSL_IVTE_Offset_An and the PSL_IVTE_Limit_An Registers. (See *Section 8.1.15* on page 95 for the mapping of an LISN to an IVTE.) The error interrupt is provided in the PSL_ErrIVTE Register. The PSL_IVTE_Offset_An and PSL_IVTE_Limit_An Registers are either loaded from the process element in the shared virtualization programming models, or set using MMIO by the hypervisor in the dedicated-process programming model.

The PSL posts an interrupt using a PCIe posted write operation to address (x'1000000000000000' | (x'00000000000' || IVTE || x'0') with data (x'0000000000000000'); where the IVTE value is calculated using the equation in *Section 8.1.15* on page 95. By performing this operation, the IVTE value directly indexes into an interrupt vector table stored in system memory.

For more information on the interrupt vector table and how interrupts are processed by the POWER processor, see the *I/O Design Architecture v2 Specification* for the Power Architecture platform.

6.1 Interrupt Types

Table 6-1 list the types of interrupts that can be received by system software and the interrupt vector table entry (IVTE) used by the PSL when presenting the interrupt. In the comment column, registers containing additional information about the interrupt are provided.

Table 6-1. Interrupt Types (Page 1 of 4)

Interrupt Type	PSL_DSISR_An												PSL_SERR_An	IVTE	Comment
	DS	DM	ST	UR	PE	AE	OC	M	P	A	S	K	HC		
Data Segment	1	0	0	0	-	-	-	-	-	-	-	-	-	IVTE0[5]	Data segment fault. The faulting EA is reported in PSL_DAR_An. The process handle for the faulting process is contained in the PSL_PEHandle_An Register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written.
Data Storage	0	1	0	0	-	-	-	1	0	0	S[1]	0	-	IVTE0[5]	Page fault or mapping fault. The faulting EA is reported in PSL_DAR_An. The process handle for the faulting process is contained in the PSL_PEHandle_An Register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register.
	0	1	0	0	-	-	-	0	P[2]	A[3]	S[1]	K[4]	-	IVTE0[5]	Protection fault or access fault. The faulting EA is reported in PSL_DAR_An. The process handle for the faulting process is contained in the PSL_PEHandle_An Register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register.

1. The PSL_DSISR_An[S] bit is set when the access causing the fault is a write type operation; otherwise, set to '0'.
2. The PSL_DSISR_An[P] bit is set when the access is not permitted by the page protection state; otherwise, set to '0'.
3. The PSL_DSISR_An[A] bit is set when a lock type operation is performed to a page marked write-through required or caching inhibited; otherwise, set to '0'.
4. The PSL_DSISR_An[K] bit is set when the access is not permitted by the virtual page class key protection; otherwise, set to '0'.
5. The value for IVTE0 is the value in PSL_IVTE_Offset_An[IVTE_Offset_0].

Table 6-1. Interrupt Types (Page 2 of 4)

Interrupt Type	PSL_DSISR_An												PSL_SERR_An	IVTE	Comment
	DS	DM	ST	UR	PE	AE	OC	M	P	A	S	K	HC		
Segment Table Pointer	0	0	1	0	-	-	-	1	0	0	0	0	-	IVTE0[5]	Page fault or mapping fault. The faulting VA is reported in SSTP0_An || SSTP1_An. The process handle for the faulting process is contained in the PSL_PEHandle_An register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register.
	0	0	1	0	-	-	-	0	P[2]	0	0	0	-	IVTE0[5]	Protection fault or access fault. The faulting VA is reported in SSTP0_An || SSTP1_An. The process handle for the faulting process is contained in the PSL_PEHandle_An Register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register. There is no key protection on the segment table pointer.
Accelerator Utilization Record	0	0	0	1	-	-	-	1	0	0	S[1]	0	-	IVTE0[5]	Page fault or mapping fault. The faulting VA is reported in AURP0_An || AURP1_An. The process handle for the faulting process is contained in the PSL_PEHandle_An register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register. There is no additional interrupt status information.
	0	0	0	1	-	-	-	0	P[2]	A[3]	S[1]	0	-	IVTE0[5]	Protection fault or access fault. The faulting VA is reported in AURP0_An || AURP1_An. The process handle for the faulting process is contained in the PSL_PEHandle_An Register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register. There is no additional interrupt status information. There is no key protection on the Accelerator Utilization Record Pointer.

1. The PSL_DSISR_An[S] bit is set when the access causing the fault is a write type operation; otherwise, set to ‘0’.
2. The PSL_DSISR_An[P] bit is set when the access is not permitted by the page protection state; otherwise, set to ‘0’.
3. The PSL_DSISR_An[A] bit is set when a lock type operation is performed to a page marked write-through required or caching inhibited; otherwise, set to ‘0’.
4. The PSL_DSISR_An[K] bit is set when the access is not permitted by the virtual page class key protection; otherwise, set to ‘0’.
5. The value for IVTE0 is the value in PSL_IVTE_Offset_An[IVTE_Offset_0].

Table 6-1. Interrupt Types (Page 3 of 4)

Interrupt Type	PSL_DSISR_An												PSL_SERR_An	IVTE	Comment
	DS	DM	ST	UR	PE	AE	OC	M	P	A	S	K	HC		
Hypervisor Context Time Warning	-	-	-	-	-	-	-	-	-	-	-	-	1	Err-IVTE_Slice	Warn hypervisor. This interrupt is caused when the AFU is not responding to a context save request and the hypervisor context warning interval has expired. The process handle for the process not responding is contained in the PSL_PEHandle_An Register. The interrupt status is located in the PSL_SERR Register. The interrupt status is reset when the corresponding interrupt status bit is written to a '1'. There is no additional status information for this interrupt.
Operating System Context Time Warning	-	-	-	-	-	-	1	-	-	-	-	-	-	IVTE0[5]	Warn operating system. This interrupt is caused when the AFU is not responding to a context save request and the operating system context warning interval has expired. The process handle for the process not responding is contained in the PSL_PEHandle_An Register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register. There is no additional interrupt status information.
PSL Slice Error	-	-	-	-	1	-	-	-	-	-	-	-	-	IVTE0[5]	PSL slice error. The PSL error interrupt type is a summary for slice errors reported by the PSL. The number and type of PSL errors is implementation specific. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register. Additional implementation-dependent interrupt status information is provided in the PSL_ErrStat_An Register.
PSL Error	-	-	-	-	-	-	-	-	-	-	-	-	-	Err-IVTE	PSL error. The PSL error interrupt type is a summary for errors reported by the PSL. The number and type of PSL errors is implementation specific. The interrupt status is located in the PSL_ErrIVTE Register. The interrupt status is reset when the corresponding interrupt status bits are written to a '1'. Additional interrupt status information is implementation dependent.

1. The PSL_DSISR_An[S] bit is set when the access causing the fault is a write type operation; otherwise, set to '0'.
2. The PSL_DSISR_An[P] bit is set when the access is not permitted by the page protection state; otherwise, set to '0'.
3. The PSL_DSISR_An[A] bit is set when a lock type operation is performed to a page marked write-through required or caching inhibited; otherwise, set to '0'.
4. The PSL_DSISR_An[K] bit is set when the access is not permitted by the virtual page class key protection; otherwise, set to '0'.
5. The value for IVTE0 is the value in PSL_IVTE_Offset_An[IVTE_Offset_0].

Table 6-1. Interrupt Types (Page 4 of 4)

Interrupt Type	PSL_DSISR_An												PSL_SERR_An	IVTE	Comment
	DS	DM	ST	UR	PE	AE	OC	M	P	A	S	K	HC		
PSL Slice Error	-	-	-	-	-	-	-	-	-	-	-	-	0	Err-IVTE_Slice	PSL error. The PSL error interrupt type is a summary for errors reported by the PSL. The number and type of PSL errors is implementation specific. The interrupt status is located in the PSL_SERR Register. The interrupt status is reset when the corresponding interrupt status bits are written to a ‘1’. Additional interrupt status information is implementation dependent.
AFU Error	-	-	-	-	-	1	-	-	-	-	-	-	-	IVTE0[5]	AFU error. An AFU error is reported by the PSL when the AFU cannot send an error to its corresponding application. The reason for the interrupt is reported in the AFU_ERR_An Register. The process handle for the process that caused the error is contained in the PSL_PEHandle_An Register. The PSL_DSISR_An status bit is reset when the corresponding PSL_TFC_An Register is written or the PSL slice is purged by writing the PSL_SCNTL_An Register. Additional interrupt status information is implementation dependent.
AFU Interrupt (level n)	-	-	-	-	-	-	-	-	-	-	-	-	-	IVTEn	AFU interrupt level n. The interrupt status and reset of interrupt status is implementation-dependent. The interrupt source number is calculated as defined in *Section 8.1.14* on page 94, *Section 8.1.15* on page 95, and *Section B.3* on page 184.

1. The PSL_DSISR_An[S] bit is set when the access causing the fault is a write type operation; otherwise, set to ‘0’.
2. The PSL_DSISR_An[P] bit is set when the access is not permitted by the page protection state; otherwise, set to ‘0’.
3. The PSL_DSISR_An[A] bit is set when a lock type operation is performed to a page marked write-through required or caching inhibited; otherwise, set to ‘0’.
4. The PSL_DSISR_An[K] bit is set when the access is not permitted by the virtual page class key protection; otherwise, set to ‘0’.
5. The value for IVTE0 is the value in PSL_IVTE_Offset_An[IVTE_Offset_0].

7. Storage Addressing

Storage addressing of the Coherent Accelerator Interface Architecture (CAIA) is compatible with the Power ISA virtual memory architecture. The memory management unit (MMU) in the PSL employs the same basic mechanism as the Power ISA to process an effective address provided by an accelerator.

To enable translations of the effective addresses used by an accelerator, set the Relocate (R) bit of the PSL State Register (PSL_SR_An) to ‘1’.

Two steps are required to convert an effective address to a real address:

1. **Convert the effective address to a virtual address.** The conversion to a virtual address uses a storage segment. The MMU in the PSL differs from the Power ISA in how the storage segments are accessed. In the Power ISA, the storage segments are provided by system software using processor instructions to load a segment lookaside buffer (SLB). For the PSL, the storage segments are in a hardware-accessed segment table located in main storage. The PSL searches the segment table for the storage segment to use for each translation. For the CAIA, the SLB is a hardware caching structure for the segment table. For both the Power ISA and the PSL, coherency of the SLB is maintained by system software.
2. **Convert the virtual address to a real address.** The conversion of a virtual address to a real address uses a page table in main storage. The page table format and the conversion process are described in *Power ISA, Book III*.

To enhance performance of effective to virtual conversions, most implementation provide a segment lookaside buffer (SLB). The SLB is, basically, a special cache for keeping the recently used segment table entries (STEs). The SLB need not be kept consistent with the hardware-accessed segment table in main storage. Coherency of the SLB is maintained by system software using the MMIO register provided by the PSL. All implementations must support an effective-to-virtual-address translation mechanism using a hardware-accessed segment table in main storage. (For more information, see *Section 7.2 Segment Lookaside Buffer Management* on page 71.)

To enhance performance of virtual-to-real conversions, most implementations provide a translation lookaside buffer (TLB). The TLB is, basically, a special cache for keeping the recently used page table entries (PTEs). The format and size of the page table is defined by the Power ISA. The TLB need not be kept consistent with the hardware-accessed page table in main storage. Coherency of the TLB is maintained by system software using instructions provided by the Power ISA. All implementations must support a virtual-to-real-address translation mechanism using a hardware-accessed page table in main storage.

7.1 Storage Segment Table

In the Power Architecture, the storage segments are loaded directly into the processor’s SLB using the **slbmte** instruction. For a CAIA-compliant accelerator, the PSL searches for a storage segment in a segment table located in system memory. The virtual address pointer to the storage segment table is defined by the concatenation of the values in the Storage Segment Table Pointer Zero Register (SSTP0_An) and Storage Segment Table Pointer One Register (SSTP1_An).

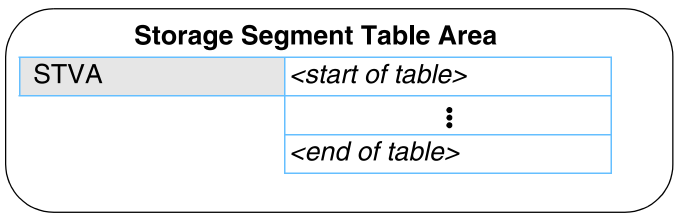
The storage segment table is a variable sized (256 B - 1 MB) structure located in main storage. The table must be naturally aligned to the size of the table, and the storage must be mapped as not caching inhibited and memory coherency required (that is a storage control setting of WIMG = ‘0010’). The real memory for the segment table must be contiguous and mapped by a single page-table entry (PTE).

Each storage segment table entry (SSTE) maps one effective segment to one virtual segment. *Table 7-1* shows the format of an entry in the storage segment table. See the *Power ISA, Book III* for a definition of each field in the SSTE.

Table 7-1. Storage Segment Table Entry

<table>
<tr><th colspan="32">Storage Segment Table Entry</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td colspan="32">Upper bits of ESID</td></tr>
<tr><td colspan="4">ESID (LSbs)</td><td>V</td><td colspan="27">Reserved</td></tr>
<tr><td colspan="2">B</td><td colspan="30">Upper bits of VSID</td></tr>
<tr><td colspan="20">Lower bits of VSID</td><td>K_s</td><td>K_p</td><td>N</td><td>L</td><td>C</td><td>Rsv</td><td colspan="2">LP</td><td colspan="4">Reserved</td></tr>
</table>

Figure 7-1. Storage Segment Table



7.1.1 Storage Segment Table Search

When an effective to virtual translation is performed, the PSL first searches for a storage segment in the storage segment table that matches the effective segment ID (ESID) of the effective address being translated. The storage segment table is indexed with p-bits where p is a function of the segment table size (p = 8 + “the number of 1’s in SSTP0[SegTableSize] field”). The search procedure is as follows:

1. **Translate the storage segment table pointer (SSTP0 || SSTP1) to a physical address:**
   The translated pointer is called the segment table origin (STABORG). To enhance performance, the PSL can keep a cached copy of the STABORG. The lower p-bits of the STABORG are always 0’s. If a matching page table entry cannot be found, a page fault is generated and the PSL_DSISR_An[ST] bit is set to a ‘1’.

   Note: The page table access for translating the segment table pointer is performed as if the PSL_SR[PR] = ‘0’. Read access to the segment table is always permitted.

2. **Primary Hash:**
   a. ***Primary Hash for a 256 MB Segment Size***
      The lower p bits of the STABORG are logically ORed with bits 43-p:35 of the effective address concatenated with ‘0000000’ (STABORG[m-p:m-1] | (EA[43-p:35] || ‘0000000’)); where m is the number of real address bits supported) to form the physical address of a group of eight SSTEs. Each of the eight SSTEs is searched to find a valid entry (V = ‘1’, B = ‘00’) that matches the ESID of the EA being translated (SSTE[ESID] = EA[0:35]). If a match is not found, a primary hash is performed for a 1 TB segment size.

   b. ***Primary Hash for a 1 TB Segment Size***
      The lower p bits of the STABORG are logically ORed with bits 31-p:23 of the effective address concatenated with ‘0000000’ (STABORG[m-p:m-1] | (EA[31-p:23] || ‘0000000’)); where m is the number of real address bits supported) to form the physical address of a group of eight SSTEs. Each of the

eight SSTEs is searched to find a valid entry (V = ‘1’, B = ‘01’) that matches the ESID of the EA being translated (SSTE[ESID] = EA[0:23]). If a match is not found, a secondary hash is performed for a 256 MB segment size if enabled (PSL_SR_An[SC] = ‘0’). If the secondary hash is disabled (PSL_SR_An[SC] = ‘1’), a segmentation fault is generated and the PSL_DSISR_An[DS] bit is set to a ‘1’.

3. **Secondary Hash:**

   a. ***Secondary Hash for a 256 MB Segment Size***
   The lower 12 bits of the STABORG are logically ORed with the one’s complement of bits 43-p:35 of the effective address concatenated with ‘0000000’ (STABORG[m-p:m-1] | (~(EA[43-p:35]) || ‘0000000’)); where m is the number of real address bits supported) to form the physical address of a group of eight SSTEs. Each of the eight SSTEs is searched to find a valid entry (V = ‘1’, B = ‘00’) that matches the ESID of the EA being translated (SSTE[ESID] = EA[0:35]). If a match is not found, a secondary hash is performed for a 1 TB segment size.

   b. ***Secondary Hash for a 1 TB Segment Size***
   The lower 12 bits of the STABORG are logically ORed with the one’s complement of bits 31-p:23 of the effective address concatenated with ‘0000000’ (STABORG[m-p:m-1] | (~(EA[31-p:23]) || ‘0000000’)); where m is the number of real address bits supported) to form the physical address of a group of eight SSTEs. Each of the eight SSTEs is searched to find a valid entry (V = ‘1’, B = ‘01’) that matches the ESID of the EA being translated (SSTE[ESID] = EA[0:23]). If a match is not found, a segmentation fault is generated and the PSL_DSISR_An[DS] bit is set to a ‘1’.

7.2 Segment Lookaside Buffer Management

The CAIA provides a set of MMIO registers to manage the SLBs in a PSL. These MMIO registers provide the same functions for a PSL MMU as the Power ISA instructions provide for a POWER processor MMU. That is, the following registers mimic the source operands of the Power ISA SLB management instructions (**slbie** and **slbia**):

- SLB Invalidate Entry Register (SLBIE_An)
- SLB Invalidate All Register (SLBIA_An)
- PSL SLB Invalidate Entry Register (PSL_SLBIE)
- PSL SLB Invalidate All Register (PSL_SLBIA)

7.3 Ordering Rules

The CAIA follows the same ordering rules as the POWER architecture. Because the POWER architecture is a weakly ordered memory model, the PSL must provide a means for an AFU to order operations.

A typical PSL implementation provides a completion status for each operation. When the AFU receives a successful completion status, the corresponding operation is guaranteed by the PSL and system to be visible to all units in the system and is ordered with any subsequent operations.

8. PSL Privileged Facilities

The registers in these sections should only be accessed by system software.

8.1 PSL Privileged 1 Facilities

The POWER Service Layer (PSL) privilege 1 facilities include the following registers:

Per AFU Slice Facilities

- PSL State Register (PSL_SR_An) (see page 74)
- PSL Logical Partition ID Register (PSL_LPID_An) (see page 77)
- PSL AFU Memory Base Address Register (PSL_AMBAR_An) (see page 78)
- PSL AFU Scratch Pad Offset Register (PSL_SPOffset_An) (see page 80)
- PSL ID Register (PSL_ID_An) (see page 82)
- PSL Slice Error Register (PSL_SERR_An) (see page 84)
- PSL Storage Description Register (PSL_SDR_An) (see page 86)
- PSL Authority Mask Override Register (PSL_AMOR_An) (see page 87)
- Hypervisor Accelerator Utilization Record Pointer Register (HAURP_An) (see page 88)
- PSL Scheduled Processes Area Pointer Register (PSL_SPAP_An) (see page 89)
- PSL Linked List Command Register (PSL_LLCMD_An) (see page 90)
- PSL Slice Control Register (PSL_SCNTL_An) (see page 91)
- PSL Context Swap Time Slice Register (PSL_CtxTime_An) (see page 93)
- PSL IVTE Offset Register (PSL_IVTE_Offset_An) (see page 94)
- PSL IVTE Limit Register (PSL_IVTE_Limit_An) (see page 95)

PSL Facilities

- PSL Context Swap Time Register (PSL_CtxTime) (see page 97)
- PSL Error Interrupt Register (PSL_ErrIVTE) (see page 99)
- PSL Key One Register (PSL_KEY1) (see page 100)
- PSL Key Two Register (PSL_KEY2) (see page 101)
- PSL Control Register (PSL_Control) (see page 102)
- AFU Download Control Register (AFU_DLCNTL) (see page 104)
- AFU Download Address Register (AFU_DLADDR) (see page 106)
- PSL Lookaside Buffer Invalidate Selection Register (PSL_LBISEL) (see page 107)
- PSL SLB Invalidate Entry Register (PSL_SLBIE) (see page 108)
- PSL SLB Invalidate All Register (PSL_SLBIA) (see page 110)
- PSL TLB Invalidate Entry (PSL_TLBIE) (see page 112)
- PSL TLB Invalidate All (PSL_TLBIA) (see page 114)
- PSL AFU Selection Register (PSL_AFUSEL) (see page 116)

8.1.1 PSL State Register (PSL_SR_An)

PSL State Register (PSL_SR_An) contains configuration information for the corresponding PSL slice.

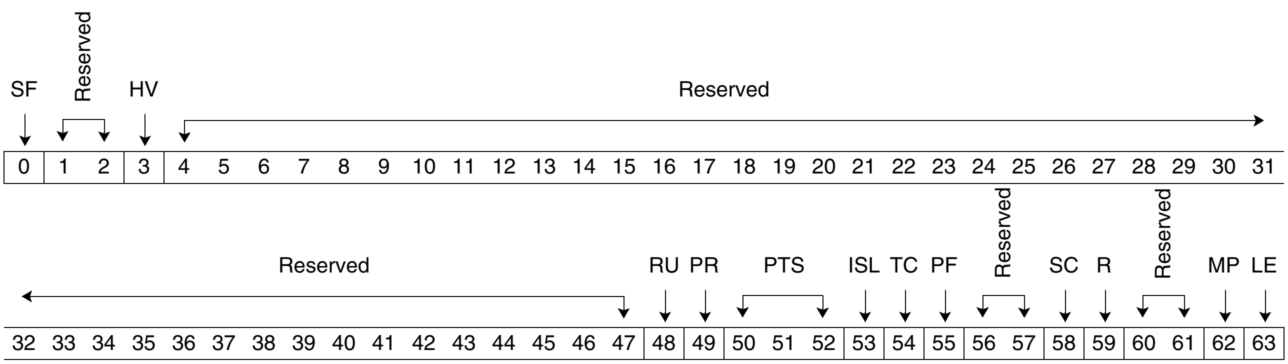
This register is initialized by the hypervisor or from the process element. The process element information is used when the PSL Scheduled Processes Area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘00’; where n is an AFU number.



Bits	Field Name	Description
0	SF	Sixty-four bit mode. 0 AFU effective addresses are interpreted as 32-bit values. 1 AFU effective addresses are interpreted as 64-bit values.
1:2	Reserved	Reserved.
3	HV	Hypervisor state. 0 AFU is not operating in a hypervisor state. 1 AFU is operating in a hypervisor state if PSL_SR_An[PR] = ‘0’; otherwise, AFU is not operating in hypervisor state. **Programming Note:** The privilege state of an AFU is determined by PSL_SR_An[HV] || PSL_SR_An[PR]. 00 AFU is operating in a privileged state. 01 AFU is operating in a problem state. 10 AFU is operating in a privileged and hypervisor state. 11 AFU is operating in a problem state.
4:47	Reserved	Reserved.

Bits	Field Name	Description
48	RU	Hot/cold reference array update control. When operating with the new PTE format enabled, a hot/cold reference array is also maintained in system memory. The PSL also sends special operations to the system to update this structure if this bit is set to ‘1’. 0 Updates of the HCA structure are not performed. 1 Updates of the HCA structure are performed.
49	PR	Problem state. 0 The PSL has privileged-state access to pages. 1 The PSL has problem-state access to pages. **Programming Note:** The PSL problem state (PR) bit is set by system software based on the use of the associated AFU. If the AFU is to be a system software resource (not under direct control of an application) and the function requires the AFU to issue memory requests with privileged-state access to pages, this bit must be cleared. If the AFU function is controlled by an application, its access must be restricted to problem state by the PR bit. The problem state control bit interacts with the Ks and Kp storage key bits in the SLB in combination with the Page Protection (PP) bits in the page table, as defined in *Power ISA, Book III*. The problem-state control is only effective in PSL translation on state (R = ‘1’).
50:52	PTS	PTE time base selection. The PTE time base selection field controls the granularity for updating the PTE time base field in the new PTE format. These bits also select which bits of the time base are used for updates of the hot/cold reference array (HCA structure in system memory). This field is only used when the new format is selected. 000 Reserved 001 0.5 s 010 1 s 011 2 s 100 4 s 101 8 s 110 16 s 111 32 s
53	ISL	Ignore segment large-page specification. When ISL mode is enabled, and address translation is enabled, and the PSL is not in hypervisor state; address translation is performed as if the contents of SLB L||LPbits are ‘000’. This also applies for virtual addresses provided in the process element. When address translation is disabled, the setting of the ISL bit has no effect. The setting of the ISL bit has no effect on translation invalidations. Hypervisor state and operating with translation disabled is not supported by the CAIA so this bit always has an effect on the address translation mechanism. 0 ISL disabled 1 ISL dnabled **Note:** This field replaces the single-step trace enable (SE) field in the Power Architecture Machine State Register (MSR) definition. For CAIA-compliant implementations, the trace facilities are controlled by a different means.
54	TC	Translation control. 0 Secondary hash of the page table search is enabled. 1 Secondary hash of the page table search is disabled. **Note:** This field replaces the branch trace enable (BE) field in the Power Architecture Machine State Register (MSR) definition. For CAIA-compliant implementations, the trace facilities are controlled by a different means.
55	PF	PTE format. This field controls the format of the page table entry (PTE). When this bit is set to ‘0’, the format is defined by the Power Architecture. When this bit is set to ‘1’, the format is implementation specific. Bits 48 and 50:52 are used as control bits for the new PTE format. 0 PTE format specified by the Power Architecture. 1 New implementation-specific PTE format.

Bits	Field Name	Description
56:57	Reserved	Reserved.
58	SC	Segment translation control. 0 Secondary hash of the segment table search is enabled. 1 Secondary hash of the segment table search is disabled. **Note:** This field replaces the Instruction Relocate (IR) field in the Power Architecture Machine State Register (MSR) definition. The CAIA does not treat instruction and data fetches from an AFU differently.
59	R	Relocate. 0 PSL translation is off. 1 PSL translation is on. **Programming Note:** The PSL Relocate (R) bit controls how effective addresses are translated into real addresses. If the relocate control specifies translation off (R = ‘0’), the effective addresses are treated as real addresses. If the relocate control is enabled (R = ‘1’), the effective addresses are translated. The SLB, TLB, and page table facilities are used to translate the effective address to a virtual address, and then to a real address.
60:61	Reserved	Reserved.
62	MP	Master process. The Master Process bit identifies the process as having authority to control and manage the resource of the AFU. This bit is provided to the AFU by the PSL. 0 The corresponding process is a user process. 1 The corresponding process is a master process and enabled to control and manage the resources of the AFU.
63	LE	Little endian. The data being accessed by the AFU is stored in a little-endian format, meaning the least significant byte first. The bit does not change the operation of the PSL. The PSL only provides this bit to the AFU in the dedicated or time-sliced programming models. The method for an AFU to determine the setting of this bit for the AFU-directed programming models is implementation dependent. 0 Big-endian format. 1 Little-endian format.

8.1.2 PSL Logical Partition ID Register (PSL_LPID_An)

The Power ISA provides a logical partitioning (LPAR) facility to permit processors and portions of main storage to be allocated to logical groups or partitions. For more information about the LPAR, see the *Power ISA, Book III*. The CAIA extends the LPAR facility to allow AFUs to also be assigned to partitions. The PSL Logical Partition ID Register (PSL_LPID_An) contains a value that identifies the partition to which an AFU is assigned.

This register is initialized by the hypervisor or from the process element. The process element information is used when the PSL Scheduled Processes Area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘08’; where n is an AFU number.

Reserved

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

LPID

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:31	Reserved	Reserved.
32:63	LPID	Logical partition ID.

8.1.3 PSL AFU Memory Base Address Register (PSL_AMBAR_An)

The PSL AFU Memory Base Address Register (PSL_AMBAR_An) contains the starting address in main storage of the AFU's memory that is mapped into system memory. The storage area defined by the PSL_AMBAR_An can be marked as cacheable storage in the hardware page table. The contents of this register is typically a subset of a larger storage region defined by the host processor's External Memory Base Address Register. The storage regions defined for all AFUs must not overlap and must be fully contained in the larger storage region defined by the host processor's External Memory Base Address Register.

This register is always initialized by the hypervisor when the AFU memory controller is enabled for the selected programming mode. The AFU memory controller model is never virtualized. This register is not valid when the AFU memory controller is not enabled for the programming model (that is, there is no AFU memory mapped into system memory).

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

This facility is optional. Reads of this register must return zeros when the AFU memory controller is not supported or not enabled for the programming model. System software can detect if this feature is supported by writing x'0FFFFFFFFFFFF03F' to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x'10'; where n is an AFU number.

E	Reserved	AMBAR
0	1 2 3	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

AMBAR	Reserved	AMSIZE
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51	52 53 54 55 56 57	58 59 60 61 62 63

Bits	Field Name	Description
0	E	Enable When this bit is set to '1', the AFU's memory is mapped into the system's address space. 0 AMBAR disabled. 1 AMBAR enabled. **Implementation Note:** If the PSL implementation does not support mapping AFU memory into the system memory address space (LPC operation), this field must be read-only and returned as a '0'. System software can read the initial value or write a '1' to this field to determine the PSL's capability to support LPC operation.
1:3	Reserved	Reserved.

Bits	Field Name	Description
4:51	AMBAR	AFU's memory origin (real address of AFU's memory that is mapped into system memory). The AMBAR field in PSL_AMBAR_An contains the high-order 48 bits of the 60-bit real address of the AFU's memory. The AFU's memory is thus constrained to lie on a 2^{12} byte (4 KB) boundary at a minimum. The number of low-order zero bits in AMBAR must be greater than or equal to the value in AMSIZE. For implementations that support a real address size of only m bits, where m is less than 62, the upper bits of the AFU's memory origin are treated as reserved bits. Software must set them to zeros.
52:57	Reserved	Reserved.
58:63	AMSIZE	Encoded size of AFU's memory. The AMSIZE field in PSL_AMBAR_An contains an integer giving the number of bits (in addition to the minimum of 12 bits) that are used to address the AFUs memory. This number must not exceed 28.

8.1.4 PSL AFU Scratch Pad Offset Register (PSL_SPOffset_An)

The PSL AFU Scratch Pad Offset Register (PSL_SPOffset_An) contains the offset in the memory attached to the PSL of the AFU's scratch pad memory. When operating in a virtualized programming model, the scratch pad can either be persistent or nonpersistent between context swaps. For the scratch pad to be persistent, the hypervisor must subdivide the adapter's memory into regions for each process. The scratch pad regions must not overlap.

This register is initialized by the hypervisor or from the process element. The process element information is used when the PSL Scheduled Processes Area is enabled (PSL_SPAP_An[V] = '1').

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

This facility is optional. Reads of this register must return zeros when the PSL AFU Scratch Pad Memory is not supported or not enabled for the programming model. System software can detect if this feature is supported by writing x'7FFFFFFFFFFFF03F' to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x'18'; where n is an AFU number.

E	P	SPOffset
0	1	2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

SPOffset	Reserved	SPSIZE
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51	52 53 54 55 56 57	58 59 60 61 62 63

Bits	Field Name	Description
0	E	Enable. When this bit is set to '1', the AFU can use the region of memory attached to the PSL defined by the SPOffset and SPSIZE fields of this register as scratch pad memory. 0 Scratch pad disabled. 1 Scratch pad enabled. **Implementation Note:** If the PSL implementation does not support a scratch pad memory for the AFU, this field must be read-only and returned as a '1'. System software can read the initial value or write a '0' to this field to determine the PSL's capability to support scratch pad memory.
1	P	Persistent. When this bit is set to '1', the contents of the AFU's scratch pad memory is preserved between context swaps in a shared programming model. 0 Scratch pad contents not preserved. 1 Scratch pad contents preserved.

Bits	Field Name	Description
2:51	SPOffset	Scratch pad offset. The SPOffset field in PSL_SPOffset_An contains the offset in the memory attached to the PSL allocated to the AFU for use as a scratch pad. The AFU's memory is constrained to lie on a 4 KB boundary. The number of low-order zero bits in SPOffset must be greater than or equal to the value in SPSIZE so that the scratch pad area is naturally aligned. The amount of memory attached to the PSL is implementation dependent. The upper address bits are ignored by the PSL.
52:57	Reserved	Reserved.
58:63	SPSIZE	Encoded size of scratch pad memory. The SPSIZE field in PSL_SPOffset_An contains an integer giving the number of bits (in addition to the minimum of 12 bits) defining the size of the AFU's scratch pad memory. The size of the scratch pad memory is $2^{(SPSIZE + 12)}$ bytes.

8.1.5 PSL ID Register (PSL_ID_An)

The PSL ID Register (PSL_ID_An) is used to define the linked list of PSLs assigned to service the scheduled processes. Contained in this register are a PSL identifier for the PSL and an identifier for the next PSL in the linked list. Along with the identifier are two flags to indicate if the PSL is the first and/or last PSL in the list.

The first PSL is responsible for monitoring the sw_command_status word in the scheduled processes area for commands requested by software. After a command is detected, the first PSL performs the requested operation and if required, forwards the command to the next PSL in the list.

PSLs in the middle of the list perform the requested command when their identifier is written to the psl_chained_command word in the scheduled processes area. It is possible in some scenarios for a PSL in the middle of the list to complete the command. In this case, the sw_command_status word in the scheduled processes area is updated by the PSL completing the command. If the command is not complete, the PSL forwards the command to the next PSL by writing the psl_chained_command word with the identifier of the next PSL in the list.

When the last PSL detects the command (either in the sw_command_status or the psl_chained_command word), the requested operation is performed and the completion status is written to the sw_command_status word in the scheduled processes area.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘20’; where n is an AFU number.

Reserved	L	F
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31

NXT_PSL_ID	PSL_ID
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:29	Reserved	Reserved.
30	L	Last PSL. The last PSL flag indicates that the PSL is the last PSL in the list of PSLs assigned to service the scheduled processes and the software commands. Software commands are not forwarded to the next PSL and the final status can be written to the *sw_command_status* word. 0 PSL is not the last PSL. 1 PSL is the last PSL. **Implementation Note:** If the PSL implementation does not support multiple PSLs operating on the same Process Element Linked List (that is, chaining PSLs), this field must be read-only and returned as a ‘1’. System software can read the initial value or write a ‘0’ to this field to determine the PSL’s capability to support chaining PSLs.

Bits	Field Name	Description
31	F	First PSL. The first PSL flag indicates that the PSL is the first PSL in the list of PSLs assigned to service the scheduled processes and the software commands. The PSL with this flag set monitors the sw_command_status word for requested commands to be performed. 0 PSL is not the first PSL. 1 PSL is the first PSL. **Implementation Note:** If the PSL implementation does not support multiple PSLs operating on the same Process Element Linked List (that is, chaining PSLs), this field must be read-only and returned as a ‘1’. System software can read the initial value or write a ‘0’ to this field to determine the PSL’s capability to support chaining PSLs.
32:47	NXT_PSL_ID	Next PSL identifier. The next PSL identifier selects which PSL assigned to service the scheduled processes must perform the operation after the current PSL has serviced the command. When the sw_command_status word is written by system software, the first PSL_ID in the list of PSLs assigned to service the processes performs the requested command. If required, the first PSL forwards the command to the next PSL in the psl_chained_command.
48:63	PSL_ID	PSL identifier. The PSL identifier is used to select which PSL assigned to service the scheduled processes must perform the operation. When the sw_command_status word is written by system software, the PSL_ID must be the first in the list of PSLs assigned to service the processes. Each PSL has the ID of the next PSL in the list and forwards the command to the next PSL in the psl_chained_command if required.

8.1.6 PSL Slice Error Register (PSL_SERR_An)

The PSL Slice Error Register (PSL_SERR_An) defines the interrupt level, which is generated when the PSL detects an error condition requiring attention from system software, typically the hypervisor. There are two errors that are reported using this interrupt level: the PSL error associated with a slice but not a particular process and the hypervisor context time warning.

Note: The PSL_IVTE_Limit_An Register is not applied to the ErrIVTE_Slice.

The PSL error conditions and status are implementation specific. For more information, see the implementation-specific documentation.

The hypervisor context time warning is generated when the AFU has not responded to a context save request from the PSL and the PSL_CtxTime[Warn_Hypervisor] time has expired.

System software should record the errors reported with this interrupt in the system error log.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

This register is optional for CAIA-compliant devices that support only a single AFU. System software can detect if this feature is supported by writing x'000000000000FFFF' to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

Note: When resetting the interrupt status bit, system software should make sure the ErrIVTE_Slice field is also written to the current interrupt vector for errors reported to the hypervisor.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x'28'; where n is an AFU number.

Implementation Dependent	AE	HC
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29	30	31

Implementation Dependent	ErrIVTE_Slice
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:29	Implementation Dependent	Reserved for implementation-dependent PSL errors. Bits in this field are set to '1' if the PSL has detected the corresponding error condition. Additional status information concerning the error might be available in an implementation-specific register. Bits in this field are reset to '0' when the this register is written with the corresponding bit set to '1'.
30	AE	Set to '1' when the AFU asserts an error in AFU directed mode. There is additional status information provided in the corresponding AFU_ERR_An. This bit is reset to '0' when this register is written with the AE bit set to '1'.
31	HC	Set to '1' if the PSL_CtxTime[Warn_Hypervisor] timer interval expires. There is no additional status information provided for this interrupt. This bit is reset to '0' when this register is written with the HC bit set to '1'.

Bits	Field Name	Description
32:47	Implementation Dependent	Reserved for implementation-dependent PSL errors. Bits in this field are set to '1' if the PSL has detected the corresponding error condition. Additional status information concerning the error might be available in an implementation-specific register. Bits in this field are reset to '0' when the this register is written with the corresponding bit set to '1'.
48:63	ErrIVTE_Slice	Error interrupt level. The field contains the interrupt source number presented when the PSL detects an error condition requiring attention from system software.

8.1.7 PSL Storage Description Register (PSL_SDR_An)

The PSL Storage Description Register (PSL_SDR_An) contains the starting address in main storage of the page table for the associated PSL and the size of the page table. The PSL_SDR_An provides the same function as the POWER Storage Description Register (SDR1). For more information about the SDR1, see *Power ISA, Book III*.

This register is initialized by the hypervisor or from the process element. The process element information is used when the PSL Scheduled Processes Area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘30’; where n is an AFU number.

Reserved	HTABORG
0 1 2 3	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

HTABORG	Reserved	HTABSIZE
32 33 34 35 36 37 38 39 40 41 42 43 44 45	46 47 48 49 50 51 52 53 54 55 56 57 58	59 60 61 62 63

Bits	Field Name	Description
0:3	Reserved	Reserved.
4:45	HTABORG	Page table origin (real address of the page table). The HTABORG field in the PSL_SDR_An Register contains the high-order 42 bits of the 60-bit real address of the page table. The page table is thus constrained to lie on a 2^{18} byte (256 KB) boundary at a minimum. The number of low-order zero bits in HTABORG must be greater than or equal to the value in HTABSIZE. For implementations that support a real address size of only m bits, where m is less than 62, the upper bits of the page table origin are treated as reserved bits. Software must set them to zeros.
46:58	Reserved	Reserved.
59:63	HTABSIZE	Encoded size of page table. The HTABSIZE field in PSL_SDR_An contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the page table index. This number must not exceed 28.

8.1.8 PSL Authority Mask Override Register (PSL_AMOR_An)

The PSL Authority Mask Override Register (PSL_AMOR_An) can be used by system software to restrict modifications of the PSL Authority Mask Register (PSL_AMR_An). This register is similar to the AMOR and UAMOR registers defined in the *Power ISA, Book III*.

The bits in the PSL_AMOR_An have a 1-to-1 correspondence with the bits in the PSL_AMR_An Register. System software must ensure that both bits corresponding to a class key in the PSL_AMR_An register contain the same value (that is, PSL_AMOR_An[(2*x)] = PSL_AMOR_An[(2*x) + 1]; where x is the class number). If this requirement is violated, the results of the PSL_AMR_An are undefined.

This register is initialized by the hypervisor for all programming models.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘38’, where n is an AFU number.

Mask

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Mask

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:63	Mask	Authority mask override. Each bit has a 1-to-1 correspondence with the bits in the PSL_AMR_An Register. Both bits corresponding to a single key must be set to the same value.

Implementation Note:

A CAIA-compliant PSL is allowed to implement a single bit corresponding to bit 0 or bit 1 of the key in the PSL_AMR_An Register. If a single bit is implemented, the PSL must return the value for both bits corresponding to the key on a read of the PSL_AMOR_An Register.

8.1.9 Hypervisor Accelerator Utilization Record Pointer Register (HAURP_An)

The Hypervisor Accelerator Utilization Record Pointer Register (HAURP_An) contains the physical address, in main storage of a 64-bit partition utilization record (HAUR). The HAUR value is an estimate of the amount of time processes running under the corresponding partition have used an accelerator function. The HAUR value is atomically adjusted by the PSL at the completion of a context time interval or when a process element is completed. The adjustment amount is implementation dependent.

This register is initialized by the hypervisor or from the process element. The process element information is used when the PSL Scheduled Processes Area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

This facility is optional. Reads of this register must return zeros when the hypervisor accelerator utilization record is not supported. System software can detect if this feature is supported by writing x‘0FFFFFFFFFFFFF80’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

The value written to the Hypervisor Accelerator Utilization Record Pointer Register (HAURP_An) is implementation specific.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘80’; where n is an AFU number.

Reserved	HAUR Real Address
0 1 2 3	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

HAUR Real Address	Reserved	V
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56	57 58 59 60 61 62	63

Bits	Field Name	Description
0:3	Reserved	Reserved.
4:56	HAUR Real Address	Real address of the hypervisor accelerator utilization record. **Note:** The lower 7-bits of the 60-bit real address pointer are always ‘0’ (that is, 128-byte aligned).
57:62	Reserved	Reserved.
63	V	When set, indicates that the Hypervisor Accelerator Utilization Record Pointer’s real address is valid. If the HAURP is valid, the PSL updates the AUR in system memory when operating in a shared programming model.

8.1.10 PSL Scheduled Processes Area Pointer Register (PSL_SPAP_An)

The PSL Scheduled Processes Area Pointer Register (PSL_SPAP_An) contains the physical address in main storage of the linked list of scheduled processes (LLSP) and the PSL queue of the process element waiting to be restarted. The amount of storage reserved for the process element queue is specified by the size field.

When the AFU is operating in a shared or AFU-directed programming model, the PSL fetches process elements from the link list pointed to by this register, if valid.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘88’; where n is an AFU number.

Reserved				SPA Real Address																											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SPA Real Address																				Size								Reserved			V
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:3	Reserved	Reserved.
4:51	SPA Real Address	Real address of the scheduled processes area. **Note:** The lower 12-bits of the 60-bit real address pointer are always ‘0’ (that is, 4 KB aligned). The scheduled processes area must also be naturally aligned to the size of the area.
52:59	Size	Size of the scheduled processes area and the corresponding number of supported process elements. ‘00000000’ 4 KB scheduled processes area; n = 24 process elements supported. ‘00000001’ 8 KB scheduled processes area; n = 54 process elements supported. ‘00000011’ 16 KB scheduled processes area; n = 114 process elements supported. ... For 32 KB, 64 KB, 128 KB, 256 KB, and 512 KB, the value for n are; 235, 476, 958, 1921, and 3849 respectively. ‘11111111’ 1 MB scheduled processes area; n = 7704 process elements supported. all other values are invalid
60:62	Reserved	Reserved.
63	V	When set, indicates that the scheduled process area’s real address is valid. If the SPA real address is not valid, the PSL does not fetch process elements from this area.

8.1.11 PSL Linked List Command Register (PSL_LLCMD_An)

The PSL Linked List Command Register (PSL_LLCMD_An) is for the management of the process elements in the scheduled processes area.

When the AFU is operating in a shared or AFU-directed programming model, the PSL fetches process elements from the link list pointed to by this register, if valid. This register is used by the hypervisor to manage the linked list.

If multiple PSLs are using the same scheduled processes area, privileged software should only issue commands to the first PSL. All other PSLs will receive commands through shared memory.

This facility is optional for CAIA-compliant devices that do not support virtualization. System software can detect if this feature is supported by writing x‘000000000000FFFF’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘90’; where n is an AFU number.

Command	Reserved
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Reserved	PE_Handle
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:15	Command	Command. x‘0000’ No command. x‘0001’ terminate_element: Terminate process element at the link provided. x‘0002’ remove_element: Remove the process element at the link provided. x‘0003’ suspend_element: Stop executing the process element at the link provided. x‘0004’ resume_element: Resume executing the process element at the link provided. x‘0005’ add_element: Software is adding a process element at the link provided. x‘0006’ update_element: Software is updating the process element state at the link provided. All other values are reserved.
16:47	Reserved	Reserved.
48:63	PE_Handle	Process element handle. The process element handle, shifted right by 7 bits, is the offset from the SPA_Base of the process element to operate on.

8.1.12 PSL Slice Control Register (PSL_SCNTL_An)

The PSL Slice Control Register (PSL_SCNTL_An) allows system software to govern the operation of a single slice of the PSL.

The Programming Model (PM) field sets the programming model for the corresponding AFU slice.

Setting the Suspend Control (Sc) bit causes the corresponding PSL slice to stop executing transactions for the corresponding AFU. Requests from the AFU might still be accepted while the PSL is suspended. To continue normal operation from a suspended state, the PSL_SCNTL_An[Sc] must be set to zero.

Setting the Purge (Pc) bit in this register causes the PSL to terminate all requests queued for the corresponding AFU. AFU requests are not accepted while the PSL is in the purge state. To continue normal operation after the purge operation, the PSL_SCNTL_An[Pc] must be set to zero.

To resume normal PSL operation:

- After a PSL hardware error, the Purge Sequence (Pc and Ps) must be issued before setting normal PSL transaction processing.
- After a PSL suspend, the Suspend Command bit (Sc) must be set to normal PSL transaction processing.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘A0’; where n is an AFU number.

Reserved	CR	PM
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14	15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Reserved	Ps	Reserved	Pc	Reserved	Ss	Reserved	Sc
32 33 34 35 36 37	38 39	40 41 42 43 44 45 46 47	48	49 50 51 52 53	54 55	56 57 58 59 60 61 62	63

Bits	Field Name	Description
0:14	Reserved	Reserved.
15	CR	Configuration required. When the CR bit is set, a configuration download of the AFU is required. This bit is set by system software at initialization or when an AFU is deallocated. This bit prevents an AFU from being enabled until an image is downloaded. 0 A download of an AFU is not required. 1 A download of an AFU is required. (Prevents the AFU from being enabled.) **Note:** This field replaces the Machine Check Interrupt Enable (ME) field in the Power Architecture Machine State Register (MSR) definition. For CAIA-compliant implementations, machine check interrupts are not sourced.
16:31	PM	Programming model type. x‘0000’ Shared: PSL-controlled time-sliced virtualization of the AFU. x‘0001’ Dedicated operating system: PSL-controlled time-sliced virtualization of the AFU. x‘0002’ Dedicated process: No AFU virtualization. x‘0004’ AFU directed-shared: AFU-controlled process element selection virtualization.
32:37	Reserved	Reserved.

Bits	Field Name	Description
38:39	Ps	PSL transaction purge status. This is a read-only field. Data written to this field is ignored. 00 Purge request not outstanding. 01 Purge of PSL transactions in process (some implementations can choose not to implement this state). 11 Purge of PSL transactions is complete.
40:47	Reserved	Reserved.
48	Pc	Purge all PSL transactions from the associated AFU. 0 No purge of PSL transactions requested. 1 Purge PSL transactions request.
49:53	Reserved	Reserved.
54:55	Ss	PSL transaction queue suspend status. This is a read-only field. Data written to this field is ignored. 00 Normal PSL transaction operation. 01 Suspend of PSL operation in process (some implementations can choose not to implement this state). 11 PSL operation suspended.
56:62	Reserved	Reserved.
63	Sc	Suspend control. Suspend PSL operation. 0 Normal PSL operation. 1 Suspend PSL operation request. **Programming Note:** The PSL Suspend control (Sc) bit allows system software to stop the PSL from processing a transfer request from the associated AFU. Pending transactions are allowed to complete. This bit is useful for handling page migration in the system.

Architecture Note:

The programming model type is broken down in several fields. The lower bits (bits 24:31) are an encode of the model type and the upper bits (bits 16:23) are enables for various features of the PSL. The model types are the various forms of virtualization: time sliced and AFU directed. Each model has two forms of virtualization: shared between partitions and dedicated to a partition. The functions enables are in the following order.

- Check LPID of PBT command – bit 22
- Check PID of PBT command – bit 21
- Lowest Point of Coherency (LPC) enable – bit 20

Mode 0x0004 is intended for XSL.

Note: This note is not intended to be an architectural statement of the bit locations. As new models and combinations are defined, the Programming Model field will be updated. Software cannot depend on the bit locations for the various features.

8.1.13 PSL Context Swap Time Slice Register (PSL_CtxTime_An)

The PSL Context Swap Time Slice Register (PSL_CtxTime_An) allows system software to govern the amount of time an AFU processes one process element. After the amount of time has expired, the PSL will request for the operation to be preempted and the context saved.

This register is initialized by the hypervisor or from the process element. The process element information is used when the PSL Scheduled Processes Area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

This facility is optional for CAIA-compliant devices that do not support virtualization. System software can detect if this feature is supported by writing x‘0000000000FF00FF’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Note: This register is provided as a place holder for storing the contents of the PSL Context Swap Time Slice Register in the Process Element. Because this register is only required for virtualized programming models and read data is intermediate, implementations can choose to not implement a read/write MMIO register for this value.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘A8’; where n is an AFU number.

Reserved
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Reserved	Idle_Time	Reserved	Context_Time
32 33 34 35 36 37 38 39	40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55	56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:39	Reserved	Reserved.
40:47	Idle_Time	Idle time. The Idle Time defines the amount of time an AFU is allowed to reach a precise state, save the context, and indicate AFU_idle before the PSL starts to increase the number of times the process element will be skipped and placed back into the PSL queue. Mapping of the value to an absolute time is implementation dependent. Setting this field to a value of x‘00’ disables the timer.
48:55	Reserved	Reserved.
56:63	Context_Time	Context swap time. The context time defines the amount of time a process element is allowed to run before the PSL sends a preempt request. Mapping of the value to an absolute time is implementation dependent. Setting this field to a value of x‘00’ disables the timer.

8.1.14 PSL IVTE Offset Register (PSL_IVTE_Offset_An)

The PSL IVTE Offset Register (PSL_IVTE_Offset_An) allows system software to program the starting offset within the interrupt vector table for 1 - 4 independent ranges of IVTEs. The size of each range is set in the PSL IVTE Limit Register (PSL_IVTE_Limit_An). See *Section 8.1.15* on page 95 for the mapping of an AFU logical interrupt source number (LISNn) to an interrupt vector table entry (IVTE).

This register is initialized by the hypervisor or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘B0’; where n is an AFU number.

IVTE_Offset_0	IVTE_Offset_1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

IVTE_Offset_2	IVTE_Offset_3
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:15	IVTE_Offset_0	IVTE offset for range 0. This field defines the starting offset for the first range of interrupts. The size of the range is set by the corresponding field in the PSL IVTE Limit Register (PSL_IVTE_Limit_An[IVTE_Range_0]).
16:31	IVTE_Offset_1	IVTE offset for range 1. This field defines the starting offset for the second range of interrupts. The size of the range is set by the corresponding field in the PSL IVTE Limit Register (PSL_IVTE_Limit_An[IVTE_Range_1]).
32:47	IVTE_Offset_2	IVTE offset for range 2 This field defines the starting offset for the third range of interrupts. The size of the range is set by the corresponding field in the PSL IVTE Limit Register (PSL_IVTE_Limit_An[IVTE_Range_2]).
48:63	IVTE_Offset_3	IVTE offset for range 3 This field defines the starting offset for the fourth range of interrupts. The size of the range is set by the corresponding field in the PSL IVTE Limit Register (PSL_IVTE_Limit_An[IVTE_Range_3]).

8.1.15 PSL IVTE Limit Register (PSL_IVTE_Limit_An)

The PSL IVTE Limit Register (PSL_IVTE_Limit_An) allows system software to govern the interrupt vector table entries (IVTEs) used by the corresponding AFU. This register allows system software to select 1 - 4 independent ranges of IVTEs. The ranges start at the corresponding interrupt vector offset (PSL_IVTE_Offset_An[IVTE_Offset_0]). The AFU's logical interrupt source number (AFU_LISN) are mapped to these ranges using the following equations. The AFU_LISN must be between in the range 1 ≤ AFU_LISN ≤ (Max_Ints - 1). The maximum number of interrupts (Max_Ints) is the sum of the interrupt ranges (PSL_IVTE_Limit_An[IVTE_Range_x]).

```
IVTE = PSL_IVTE_Offset_An[IVTE_Offset_0] + AFU_LISN;
          when  (1 ≤ AFU_LISN < PSL_IVTE_Limit_An[IVTE_Range_0])
          AND   (PSL_IVTE_Limit_An[IVTE_Range_0] != 0)
       else

IVTE = PSL_IVTE_Offset_An[IVTE_Offset_1] + (AFU_LISN - PSL_IVTE_Limit_An[IVTE_Range_0]);
          when  ( (PSL_IVTE_Limit_An[IVTE_Range_0])
                ≤ AFU_LISN <
                  (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1]) )
          AND   (PSL_IVTE_Limit_An[IVTE_Range_1] != 0)
       else

IVTE = PSL_IVTE_Offset_An[IVTE_Offset_2] + ( AFU_LISN -
       (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1]) );
          when  ( (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1])
                ≤ AFU_LISN <
                (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1] +
                 PSL_IVTE_Limit_An[IVTE_Range_2]) )
          AND   (PSL_IVTE_Limit_An[IVTE_Range_2] != 0)
       else

IVTE = PSL_IVTE_Offset_An[IVTE_Offset_3] + ( AFU_LISN -
       (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1] +
       PSL_IVTE_Limit_An[IVTE_Range_2]) );
          when  ( (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1] +
                   PSL_IVTE_Limit_An[IVTE_Range_2])
                ≤ AFU_LISN <
                  (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1] +
                   PSL_IVTE_Limit_An[IVTE_Range_2] + PSL_IVTE_Limit_An[IVTE_Range_3]) )
          AND   (PSL_IVTE_Limit_An[IVTE_Range_3] != 0)
       else

No IVTE is generated and No Interrupt is sent for the AFU.
```

Note: For PSL interrupts reported in the PSL_DSISR_An Register, the IVTE is PSL_IVTE_Offset_An[IVTE_Offset_0] (a LISN of zero). For PSL error interrupts, the IVTE is sourced directly from the PSL_ErrIVTE register. The AFU_LISN must be between in the range 1 ≤ AFU_LISN < (Max_Ints - 1). The maximum number of interrupts (Max_Ints) is the sum of the interrupt ranges (PSL_IVTE_Limit_An[IVTE_Range_x]).

This register is initialized by the hypervisor or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = '1').

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P1_Base | P1(n)) + x‘B8’; where n is an AFU number.

Int_Range_0	Int_Range_1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Int_Range_2	Int_Range_3
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:15	Int_Range_0	Interrupt range 0. The Int_Range_0 defines the number of interrupts within the range starting at the corresponding interrupt vector (PSL_IVTE_An[IVTE_Offset_0]). Int_Range_Size_0 must be greater than or equal to 1. The PSL uses the first interrupt in this range (PSL_IVTE_An[IVTE_Offset_0]) for translation faults and errors reported using the PSL_DSISR_An Register. System software can write this field to x‘FFFF’ and read back the maximum range value supported by the PSL.
16:31	Int_Range_1	Interrupt range 1. The Int_Range_S1 defines the number of interrupts within the range starting at the corresponding interrupt vector (PSL_IVTE_An[IVTE_Offset_1]). A size of zero disables this interrupt range. System software can write this field to x‘FFFF’ and read back the maximum range value supported by the PSL.
32:47	Int_Range_2	Interrupt range 2 The Int_Range_2 defines the number of interrupts within the range starting at the corresponding interrupt vector (PSL_IVTE_An[IVTE_Offset_2]). A size of zero disables this interrupt range. System software can write this field to x‘FFFF’ and read back the maximum range value supported by the PSL.
48:63	Int_Range_3	Interrupt range 3. The Int_Range_3 defines the number of interrupts within the range starting at the corresponding interrupt vector (PSL_IVTE_An[IVTE_Offset_3]). A size of zero disables this interrupt range. System software can write this field to x‘FFFF’ and read back the maximum range value supported by the PSL.

8.1.16 PSL Context Swap Time Register (PSL_CtxTime)

The PSL Context Swap Time Register (PSL_CtxTime) allows system software to define the amount of time an AFU is allowed to continue processing a job before the PSL requests that the operation be preempted and the context saved.

A timer is started when a context swap is requested by the PSL. The timer is stopped when the AFU responds to the context swap request. If the timer exceeds the Warn_OS time before the AFU responds, a data storage interrupt is presented to the operating system; logical interrupt source number 0 (LISN0). The actions taken by the operating system is implementation specific.

The Warn_OS time field is only used for time-sliced virtualization programming models.

A timer is started when an interrupt is presented to the operating system for exceeding the Warn_OS time in a time-sliced virtualization model, or when any interrupt is presented in a non time-sliced virtualization programming model. The timer is stopped when either the AFU responds to the context swap request in a time-sliced virtualization programming model or when the corresponding PSL_TFC_An Register is written in a non time-sliced programming model. If the timer exceeds the Warn_Hypervisor time, a PSL Error (ErrIVTE) is presented to the hypervisor. The actions taken by the hypervisor is implementation specific.

The content this register should be initialized by system software.

This facility is optional for CAIA-compliant devices that do not support virtualization. System software can detect if this feature is supported by writing x‘00FF00FF00000000’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset P1_Base + x‘0000’

Reserved	Warn_Hypervisor	Reserved	Warn_OS
0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31

Reserved
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:7	Reserved	Reserved.
8:15	Warn_Hypervisor	Warn hypervisor time. The warn hypervisor time defines the amount of time after the warning is sent to the operating system when the PSL sends an interrupt to the hypervisor. After the interrupt is presented to the AFU, the context of the AFU is not swapped until the interrupt is acknowledged. The actions taken by the hypervisor are implementation dependent. Mapping of the value to an absolute time is implementation dependent. Setting this field to a value of x‘00’ disables the timer.
16:23	Reserved	Reserved.

Bits	Field Name	Description
24:31	Warn_OS	Warn OS time. The warn OS time defines the amount of time the AFU is allowed to complete a context save after being preempted by the PSL. When the warn OS time is exceeded, the PSL sends an interrupt to the OS. After the interrupt is presented to the AFU, the context of the AFU is not swapped until the interrupt is acknowledged. The actions taken by the OS is operating-system dependent. See the operating system manual for more information. Mapping of the value to an absolute time is implementation dependent. Setting this field to a value of x‘00’ disables the timer.
32:63	Reserved	Reserved.

8.1.17 PSL Error Interrupt Register (PSL_ErrIVTE)

The PSL Error Interrupt Register (PSL_ErrIVTE) defines the interrupt level, which is generated when the PSL detects an error condition requiring attention from system software, typically the hypervisor. The PSL errors, that are reported using this interrupt level are errors that are not associated with any particular process or AFU slice.

Note: The PSL_IVTE_Limit_An Register is not applied to the ErrIVTE.

The PSL error conditions and status are implementation specific. For more information, see the implementation-specific documentation.

System software should record the errors reported with this interrupt in the system error log.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Note: When resetting the interrupt status bit, system software should make sure the ErrIVTE field is also written to the current interrupt vector for errors reported to the hypervisor.

Access Type Read/Write

Base Address Offset P1_Base + x'0008'

Implementation Dependent	TE
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	31

Implementation Mask	ErrIVTE
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:30	Implementation Dependent	Reserved for implementation-dependent PSL errors. Bits in this field are set to '1' if the PSL has detected the corresponding error condition. Additional status information about the error might be available in an implementation-specific register. Bits in this field are reset to '0' when the register is written with the corresponding bit set to '1'.
31	TE	Time-base error. This bit is set with the PSL receives an error response when updating the time-base value. This bit is reset to '0' when this register is written with the TE bit set to '1'. After an error, time-base synchronization is stopped. The following procedure must be initiated to restart time-base synchronization. 1. Clear the time-base error by writing PSL_ErrIVTE[TE] to '1'. 2. Disable time-base synchronization by writing PSL_Control[TB] to '0'. 3. Clear the error and restart time-base synchronization in the host. (See the appropriate system specifications for the procedure.) 4. Enable time-base synchronization by writing PSL_Control[TB] to '1'.
32:47	Implementation Mask	Reserved for an implementation-dependent PSL error that can be masked. Bits in the fields are set to '0' if system software wants to receive an interrupt with the associated error bits [0:15]. Setting a bit in this field to '1' masks the error and an interrupt is not presented by the PSL.
48:63	ErrIVTE	Error interrupt level. The field contains the interrupt source number presented when the PSL detects an error condition requiring attention from system software.

8.1.18 PSL Key One Register (PSL_KEY1)

The PSL Key One Register (PSL_KEY1) is used by system software to authenticate the PSL design. The value written to this register is implementation specific.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Access Type Write

Base Address Offset P1_Base + x'0010'

KEY1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

KEY1

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:63	Key1	PSL key one. The key value written to this register is used by system software as an authentication challenge of the PSL. The actions taken by the PSL when this register is written are implementation specific.

8.1.19 PSL Key Two Register (PSL_KEY2)

The PSL Key Two Register (PSL_KEY2) is used by system software to authenticate the PSL design. The value written to this register is implementation specific.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Access Type Write

Base Address Offset P1_Base + x‘0018’

KEY2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

KEY2

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:63	Key2	PSL key two. The key value written to this register is used by system software as an authentication challenge of the PSL. The actions taken by the PSL when this register is written are implementation specific.

8.1.20 PSL Control Register (PSL_Control)

The PSL Control Register (PSL_Control) contains general control settings for the overall PSL.

Setting the PSL cache flush request (Fr) bit causes the PSL to flush the data from the cache. For proper operation, the AFUs should be stopped before issuing this request to avoid new data from being read back into the cache. To continue normal operation from a suspended state, the PSL_Control[Fr] must be set to zero.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset P1_Base + x‘0020’

Reserved	Fs	Reserved	Fr
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27	28 29	30	31

Reserved	TB
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62	63

Bits	Field Name	Description
0:27	Reserved	Reserved
28:29	Fs	PSL cache flush status. This is a read-only field. Data written to this field is ignored. 00 Normal PSL operation. 01 PSL cache operations in process (some implementations can choose not to implement this state). 11 PSL cache flush operation complete.
30	Reserved	Reserved.
31	Fr	PSL cache flush request. 0 Normal PSL operation. 1 PSL cache flush operation request. **Programming Note:** The PSL cache flush request (Fr) bit provides system software the ability to flush the PSL’s cache in a single operation. For proper operation, all the AFUs should be stopped before issuing this request. Any pending transactions are allowed to complete.
32:62	Reserved	Reserved.

<table>
<tr><th>Bits</th><th>Field Name</th><th>Description</th></tr>
<tr><td>63</td><td>TB</td><td>Time-base enable.
Enable the synchronization of the PSL’s time base with the host system.
0 Time-base synchronization disabled.
1 Time-base synchronization enabled.

Programming Note:

Changing the TB bit from ‘0’ to ‘1’ causes the PSL to start the synchronization of the PSL’s local time base with the time base of the host system’s time base. The time-base synchronization process continues to keep the PSL’s local time base synchronized with the host system’s time base as long as the TB bit is set to ‘1’. Changing the TB bit from ‘1’ to ‘0’ causes the PSL to stop the synchronization process.
The host system must be enabled for the synchronization process before setting the TB bit.</td></tr>
</table>

8.1.21 AFU Download Control Register (AFU_DLCNTL)

The AFU Download Control Register (AFU_DLCNTL) is used to control the configuration of an AFU.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

This register is optional for CAIA-compliant devices that do not support loadable AFUs. System software can detect if this feature is supported by writing x‘00000000000FFFFF’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

Access Type Read/Write

Base Address Offset P1_Base + x‘0060’

Reserved																												D	C	E	S
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Reserved												AFU_DLSize																			
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:27	Reserved	Reserved.
28	D	Configuration done (read-only). This bit is set when the internal reconfiguration controller has received a complete image for the AFU. 0 AFU download is not complete. 1 AFU download is complete.
29	C	CRC error (read-only). This bit is set when the internal reconfiguration controller has detected an error during or after the configuration of the AFU. 0 No CRC error was detected. 1 A CRC error was detected while downloading or after an AFU configuration.
30	E	Configuration error (read-only). This bit is set when the internal reconfiguration controller has detected an error during the configuration of the AFU. 0 No error was detected. 1 An error was detected while downloading an AFU configuration.
31	S	Start download. Setting this bit to a ‘1’ starts the download of the AFU configuration information from system memory. This bit is reset by hardware when the operation is complete. Software must never write this bit to zero while a download is in process unless aborting the download sequence. Setting this bit to zero terminates the current download sequence and resets the download process. 0 AFU configuration download not in process. 1 AFU configuration download in process.
32:43	Reserved	Reserved.

Bits	Field Name	Description
44:63	AFU_DLSize	AFU download size. The AFU download size field specifies the amount of data, in blocks of 128 bytes, to read from system memory and send to the AFU configuration controller. Multiple downloads might be required to completely configure an AFU. On a read, this field contains the amount of data, in blocks of 128 bytes, remaining in the transfer.

8.1.21.1 AFU Download Procedure

The procedure for downloading an AFU is as follows:

1. Reset the AFU to make sure no operations are pending, if required.
   a. Write the AFU_CNTL_An[RA] = ‘1’
   b. Wait for AFU_CNTL_An[RS] = ‘10’
2. Purge the PSL. (Only needed if the AFU did not cleanly exit)
   a. Write the PSL_SCNTL_An[Pc] = ‘1’
   b. Wait for PSL_SCNTL_An[Ps] = ‘11’
3. Set the configuration required (PSL_SCNTL_An[CR]) bit if not already set.
4. Write the real address of the first image block to the AFU_DLADDR Register.
5. Write the block size and set the start download bit (AFU_DLCNTL[S] = ‘1’) to start the download.
6. Read the AFU_DLCNTL Register to determine when the block is complete and the device is ready for the next block (AFU_DLCNTL[S] = ‘0’), when the AFU download is done (AFU_DLCNTL[D] = ‘1’), or if an error has occurred during the download (AFU_DLCNTL[E] = ‘1’ or AFU_DLCNTL[C] = ‘1’). If an error has occurred, terminate the AFU download procedure and enter error recovery or retry the AFU download. If no error has occurred and the current block is complete, continue with the next step.
7. Write the real address of the next image block to the AFU_DLADDR register then go to step 5.

8.1.22 AFU Download Address Register (AFU_DLADDR)

The AFU Download Address Register (AFU_DLADDR) is used to control the configuration of an AFU.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

This register is optional for CAIA-compliant devices that do not support loadable AFUs. System software can detect if this feature is supported by writing x‘0FFFFFFFFFFFFF80’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

Access Type Read/Write

Base Address Offset P1_Base + x‘0068’

Reserved	AFU Image Real Address
0 1 2 3	4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

AFU Image Real Address	Reserved
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56	57 58 59 60 61 62 63

Bits	Field Name	Description
0:3	Reserved	Reserved.
4:56	AFU Image Real Address	Real address of the AFU image. **Note:** The lower 7-bits of the 60-bit real address pointer are always ‘0’ (that is, 128-byte aligned). The image data must also start on the same 128-byte alignment.
57:63	Reserved	Reserved.

8.1.23 PSL Lookaside Buffer Invalidate Selection Register (PSL_LBISEL)

PSL Lookaside Buffer Invalidate Selection Register (PSL_LBISEL) contains the process identifier (PID) and the logical partition ID (LPID) used to select which SLB and TLB entries to invalidate.

This register is typically used by the hypervisor when the AFU is operating in a shared programming mode.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset P1_Base + x‘0080’

PID

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

LPID

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:31	PID	Process identifier (PID). The PID field is used to selectively invalidate only the SLB entries for the specified process ID that match the ESID field.
32:63	LPID	Logical partition ID (LPID). The LPID field is used to selectively invalidate only the SLB entries for the specified logical partition that match the ESID field.

8.1.24 PSL SLB Invalidate Entry Register (PSL_SLBIE)

A write to the PSL SLB Invalidate Entry Register (PSL_SLBIE) causes the Valid (V) bit in all entries of the SLB that match the ESID, class, and segment size to be set to ‘0’. For any value other than ‘00’ in the invalidation qualifier (IQ) field, the PSL_LBISEL Register must be written before writing the PSL_SLBIE Register.

When an AFU is in a shared programming mode, the operating system does not have direct access to the privileged 2 SLB Invalidate Entry Register (SLBIE_An) facility to invalidate segment translations. In this case, the operating systems request the hypervisor to perform the segment translation invalidations on behalf of operating system using the privileged 1 PSL SLB Invalidate Entry Register (PSL_SLBIE) facility.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Address Write

Base Address Offset P1_Base + x‘0088’

Register Write Field Description:

ESID
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

ESID	C	SS	Reserved	IQ
32 33 34 35	36	37 38	39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61	62 63

Bits	Field Name	Description
0:35	ESID	Effective segment ID.
36	C	Class. The class field is used in conjunction with the PSL SLB Invalidate Entry Register (PSL_SLBIE). It is used as an additional qualifier for the ESID when multiple virtual address spaces exist.
37:38	SS	Segment size. The segment size field is used in conjunction with the PSL SLB Invalidate Entry Register (PSL_SLBIE). It is used as an additional qualifier for the ESID when multiple virtual address spaces exist.
39:61	Reserved	Set to zeros.
62:63	IQ	SLB invalidation qualifier (IQ). The IQ field is used to selectively invalidate only the SLB entries based on the (ESID, ESID, and LPID) or (ESID, PID, and LPID) combination. The value for the PID and LPID is defined by the PSL_LBISEL Registers. 00 Invalidate the SLBs matching the ESID. 01 Invalidate the SLBs matching the ESID and LPID. 11 Invalidate the SLBs matching the ESID, LPID, and PID.

Register Read Field Description:

Reserved																								Max_SLBIEs							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Reserved																								SLBIEs_Pending							
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:23	Reserved	Set to zeros.
24:31	Max_SLBIEs	Maximum number of SLBIE commands supported. This field indicates the maximum number of outstanding SLB invalidate entry commands supported. This value of this field is implementation dependent.
32:55	Reserved	Set to zeros.
56:63	SLBIEs_Pending	Number of SLBIE command pending. The SLBIEs_Pending field indicates the number of SLBIE commands currently outstanding. This field is used to determine when the previously issued SLB invalidations are complete. Issuing any additional SLB invalidates (that is, writing this register) when the number of invalidation pending is at the maximum can result in an SLB invalidate being lost or discarded. x‘00’ No SLB invalidate entry commands are pending. x‘01’ One SLB invalidate entry command is pending. ... x‘FF’ 255 SLB invalidateentry commands still pending.

8.1.25 PSL SLB Invalidate All Register (PSL_SLBIA)

A write to the PSL SLB Invalidate All Register (PSL_SLBIA) causes the valid (V) bit in all entries of the SLB to be set to ‘0’, making the entries invalid. The remaining fields of each entry are undefined. For any value other than ‘00’ in the invalidation qualifier (IQ) field, the PSL_LBISEL Register must be written before writing the PSL_SLBIA Register.

When an AFU is in a shared programming mode, the operating system does not have direct access to the privileged 2 SLB Invalidate All Register (SLBIA_An) facility to invalidate segment translations. In this case, the operating systems request the hypervisor to perform the segment translations invalidations on behalf of the operating system using the privileged 1 PSL SLB Invalidate All Register (PSL_SLBIA) facility.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Access Type Write

Base Address Offset P1_Base + x‘0090’

Register Write Field Description:

Field	Bits
Reserved	0–31
Reserved	32–58
IH	59–61
IQ	62–63

Bits	Field Name	Description
0:58	Reserved	Set to zeros.
59:61	IH	SLB invalidation hint (IH). The IH field is provided by system software as a hint that can be used to selectively invalidate entries in the implementation-specific look aside information. 000 Invalidate all implementation-specific lookaside information. (This is not a hint.) 001 Preserve implementation-specific lookaside information having a class value of ‘0’. 010 Preserve implementation-specific lookaside information created when MSR[IR/DR] = ‘0’. (This encoding is not valid for PSL implementations. Specific to the Power Architecture.) 110 Preserve implementation-specific lookaside information created when MSR[HV] = ‘1’, MSR[PR] = ‘0’, and MSR[IR/DR] = ‘0’. (This encoding is not valid for PSL implementations. Specific to the Power Architecture.) Other All other IH values are reserved.
62:63	IQ	SLB invalidation qualifier (IQ). The IQ field is used to selectively invalidate only the SLB entries based on the “LPID” or “PID and LPID” combination. The value for the PID and LPID is defined by the PSL_LBISEL Register. 00 Invalidate all SLBs. 01 Invalidate the SLBs matching the LPID. 11 Invalidate the SLBs matching the LPID and PID.

Register Read Field Description:

Reserved																															1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Reserved																															P
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:30	Reserved	Set to zeros.
31	Reserved	Set to one. This field indicates the maximum number of outstanding SLB invalidate all commands supported. This field is set to ‘1’ for TLB invalidate all commands.
32:62	Reserved	Set to zeros.
63	P	SLB invalidations pending. The SLB invalidation pending (P) field is used to determine when the previously issued SLB invalidations are complete. Issuing any additional SLB invalidates (that is, writing this register) when the number of invalidation pending is at the maximum may result in a SLB Invalidate being lost or disguarded. 0 No SLB invalidate all commands are pending. 1 A SLB invalidate all command is pending.

Implementation Note:

The PSL SLB Invalidate All Register (PSL_SLBIA) for a PSL MMU clears the V bit of SLB entry 0. This differs from the Power ISA **slbia** instruction, which does not clear the V bit of SLB entry 0. The PSL should fully decode the IH field. Only encodes of ‘000’ and ‘001’ for the IH field are valid for PSL implementations. Hints involving when the implementation-specific lookaside information is created based on the MSR value is not known to the PSL.

8.1.26 PSL TLB Invalidate Entry (PSL_TLBIE)

A write to the PSL TLB Invalidate Entry (PSL_TLBIE) causes the Valid (V) bit in the matching entries of the TLB to be set to ‘0’, making the entries invalid. The remaining fields of each entry are undefined. For any value other than ‘00’ in the Invalidation Qualifier (IQ) field, the PSL_LBISEL Register must be written before writing the PSL_TLBIE Register.

This register is typically used by the hypervisor when TLB Invalidate instructions are not broadcast to all processing elements in the system. This register is similar to the *TLB Invalidate Entry Local* (POWER processor instruction).

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset P1_Base + x‘00A0’

Register Write Field Description:

Field	Bits
AVA	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

AVA	AVA/LP	IQ	B	AP/AVAL	L
32 33 34 35 36 37 38 39 40 41 42 43	44 45 46 47 48 49 50 51	52 53	54 55	56 57 58 59 60 61 62	63

Bits	Field Name	Description
0:43	AVA	Abbreviated virtual address. This field contains bits 14:57 of the virtual address translated by the TLB to be invalidated. For more information on the details of this field, see the *Power Architecture Book III*.
44:51	AVA/LP	Abbreviated virtual address (AVA) when PSL_TLBIE[L] = ‘0’. When the L bit is set to zero, this field contains bits 58:65 of the virtual address translated by the TLB to be invalidated. Large Page Size Selector (LP) when PSL_TLBIE[L] = ‘1’. When the L bit is set to one, this field contains the Large Page size selector.The size of the base page and actual page size selected by the LP field is implementation dependent. The number of concurrent page sizes supported is also implementation dependent. The bits represented by “z” in this field specify the base page size and actual page size. The bits represented by “r” are the lower bits of the virtual address. Depending on the page size selected, the bits in this field represented by “r” might be concatenated with the VPN field to determine which entries are invalidated. `rrrrrrrr` TLB_Invalidate_Entry[L] = ‘0’. `rrrrrrrz` If TLB_Invalidate_Entry[L] = '1', actual page size ≥ 8 KB. `rrrrrrzz` If TLB_Invalidate_Entry[L] = '1', actual page size ≥ 16 KB. ... `zzzzzzzz` If TLB_Invalidate_Entry[L] = '1', actual page size ≥ 1 MB. Software should set the least-significant bit of the LP field to the same value as the LS bit for compatibility with implementations that only support two large page sizes. For more information on the details of this field, see the *Power Architecture Book III*.

Bits	Field Name	Description
52:53	IQ	TLB invalidation qualifier (IQ). The IQ field is used to selectively invalidate only the TLB entries based on the “LPID” or “PID and LPID” combination. The value for the PID and LPID is defined by the PSL_LBISEL Registers. 00 Invalidate all TLBs. 01 Invalidate the TLBs matching the LPID. 11 Invalidate the TLBs matching the LPID and PID. **Note:** This field replaces the IQ field in the Power Architecture for a **tlbiel** instruction. For CAIA-compliant implementations, the IQ field is assumed to always be ‘00’.
54:55	B	Segment size selector. The segment size selector defines the size of the segment used for the TLB. The segment size selector field must match the TLB entry being invalidated.
56:62	AP/AVAL	Actual page size (AP) when PSL_TLBIE[L] = ‘0’. When the L bit is set to zero, the base page size is 4 KB and bits 56:58 specify the actual page size of the TLB entry to be invalidated. System software must set bits 56:58 to the encoding ($SLBE_{L\|\|LP}$) for the actual page size specified by the PTE. Abbreviated virtual address, lower (AVAL) when PSL_TLBIE[L] = ‘1’. When the L bit is set to one, this field must contain the lower bits (starting with bit 58) of the virtual address translated by the TLB to be invalidated. The number of bits used by the hardware in this field is determined by the base page size. For more information on the details of this field, see the *Power Architecture Book III*.
63	L	Large page indicator 0 Page is small (4 KB). 1 Page is large. See LP field.

Register Read Field Description:

Reserved	Max_TLBIEs
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31

Reserved	TLBIEs_Pending
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55	56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:23	Reserved	Set to zeros.
24:31	Max_TLBIEs	Maximum number of TLBIE commands supported. This field indicates the maximum number of outstanding TLB invalidate entry commands supported. This value of this field is implementation-dependent.
32:55	Reserved	Set to zeros.
56:63	TLBIEs_Pending	Number of TLBIE commands pending. The TLBIEs_Pending field indicates the number of TLBIE commands currently outstanding. This field is used to determine when the previously issued TLB invalidations are complete. Issuing any additional TLB invalidates (that is, writing this register) when the number of invalidation pending is at the maximum can result in a TLB invalidate being lost or discarded. x‘00’ No TLB invalidate entry commands are pending. x‘01’ One TLB invalidate entry command is pending. ... x‘FF’ 255 TLB invalidate entry commands still pending.

8.1.27 PSL TLB Invalidate All (PSL_TLBIA)

A write to the PSL TLB Invalidate All (PSL_TLBIA) causes the valid (V) bit in all entries of the TLB to be set to ‘0’, making the entries invalid. The remaining fields of each entry are undefined. For any value other than ‘00’ in the invalidation qualifier (IQ) field, the PSL_LBISEL Register must be written before writing the PSL_TLBIA Register.

This register is typically used by the hypervisor when TLB invalidate instructions are not broadcast to all processing elements in the system. This register is similar to the TLB Invalidate All (**tlbia**) POWER processor instruction.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset P1_Base + x‘00A8’

Register Write Field Description:

Field	Bits
Reserved	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Reserved	32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61
IQ	62 63

Bits	Field Name	Description
0:61	Reserved	Set to zeros.
62:63	IQ	TLB invalidation qualifier (IQ). The IQ field is used to selectively invalidate only the TLB entries based on the “LPID” or “PID and LPID” combination. The value for the PID and LPID is defined by the PSL_LBISEL Registers. 00 Invalidate all TLBs. 01 Invalidate the TLBs matching the LPID. 11 Invalidate the TLBs matching the LPID and PID.

Register Read Field Description:

Reserved (0:30)	1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30	31

Reserved (32:62)	P
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62	63

Bits	Field Name	Description
0:30	Reserved	Set to zeros.
31	Reserved	Set to one. This field indicates the maximum number of outstanding TLB invalidate all commands supported. This field is set to one for TLB invalidate all commands.
32:62	Reserved	Set to zeros.
63	P	TLB invalidations pending. The TLB invalidation pending (P) field is used to determine when the previously issued TLB invalidations are complete. Issuing any additional TLB invalidates (that is, writing this register) when the number of invalidation pending is at the maximum can result in a TLB invalidate being lost or disguarded. 0 No TLB invalidate all commands are pending. 1 A TLB invalidate all command is pending.

8.1.28 PSL AFU Selection Register (PSL_AFUSEL)

The PSL AFU Selection Register (PSL_AFUSEL) contains the AFU slice number used to select which SLB and TLB entries to invalidate. This register provides system software the control to invalidate SLB and TLB entries for a specific AFU or all AFUs.

This register is typically used by the hypervisor when the AFU is operating in a shared programming mode.

There is only one register for the PSL. Access to this register should be privileged. This register must be accessed using a single 64-bit store operation.

This register is optional for CAIA-compliant devices that support only a single AFU. System software can detect if this feature is supported by writing x‘00000000000001FF’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

Access Type Read/Write

Base Address Offset P1_Base + x‘00B0’

Reserved

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Reserved | A | AFU_Slice

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:54	Reserved	Reserved.
55	A	Select all AFU slices. 0 SLB and TLB invalidate commands are performed on the AFU slice selected by PSL_AFUSEL[AFU_Slice]. 1 SLB and TLB invalidate commands are performed on all AFU slices.
56:63	AFU_Slice	AFU slice. This field contains the AFU slice number, the PSL_SLBIE, PSL_SLBIA, PSL_TLBIE, and PSL_TLBIA commands target. If the PSL_AFUSEL[A] bit is ‘1’, this field is not used and the commands target all AFU slices.

8.2 PSL Privileged 2 Facilities

The registers in this section should only be accessed by system software. The POWER Service Layer (PSL) privilege 2 facilities include the following registers:

- PSL Process and Thread Identification Register (PSL_PID_TID_An) (see page 118)
- Context Save/Restore Pointer Register (CSRP_An) (see page 119)
- Accelerator Utilization Record Pointer Zero Register (AURP0_An) (see page 120)
- Accelerator Utilization Record Pointer One Register (AURP1_An) (see page 121)
- Storage Segment Table Pointer Zero Register (SSTP0_An) (see page 123)
- Storage Segment Table Pointer One Register (SSTP1_An) (see page 125)
- PSL Authority Mask Register (PSL_AMR_An) (see page 126)
- SLB Invalidate Entry Register (SLBIE_An) (see page 128)
- SLB Invalidate All Register (SLBIA_An) (see page 130)
- SLB Invalidate Selection Register (SLBI_Select_An) (see page 132)
- PSL Data Storage Interrupt Status Register (PSL_DSISR_An) (see page 133)
- PSL Data Address Register (PSL_DAR_An) (see page 134)
- PSL Data Segment Register (PSL_DSR_An) (see page 135)
- PSL Translation Fault Control Register (PSL_TFC_An) (see page 136)
- PSL Process Element Handle Register (PSL_PEHandle_An) (see page 138)
- PSL Error Status Register (PSL_ErrStat_An) (see page 139)
- AFU Control Register (AFU_Cntl_An) (see page 140)
- AFU Error Register (AFU_ERR_An) (see page 142)
- PSL WED Register (PSL_WED_An) (see page 143)

8.2.1 PSL Process and Thread Identification Register (PSL_PID_TID_An)

The PSL Process and Thread Identification Register (PSL_PID_TID_An) contains the current process identifier (PID) and the current thread identifier (TID).

This register is initialized by the operating system or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0000’; where n is an AFU number.

TID

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

PID

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:31	TID	Thread identifier.
32:63	PID	Process identifier.

8.2.2 Context Save/Restore Pointer Register (CSRP_An)

The Context Save/Restore Pointer Register (CSRP_An) contains the effective address of a buffer in main storage for the AFU's context data. The buffer is located in the application's effective address space.

This register is initialized by the operating system or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = '1').

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

This facility is optional. System software can detect if this feature is supported by writing x'FFFFFFFFFFFFFFFF' to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

Note: This register is provided as a place holder for storing the contents of the context save/restore pointer in the process element. Because this register is only required for virtualized programming models and read data is intermediate, implementations might choose to not implement a read/write MMIO register for this value.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x'0008'; where n is an AFU number.

Context Save / Restore Effective Address

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Context Save / Restore Effective Address | Limit

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 | 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:51	Effective Address	Effective address of the application's context save/restore area. **Note:** The lower 12-bits of the 64-bit application context save/restore real address are always '0' (that is, 4 KB aligned). In addition, the EA must be aligned on a limit boundary.
52:63	Limit	Context save/restore area limit. The 12-bit limit field is used to define the number of pages minus '1' in the application context save/restore area. The last byte of the area is (CSRP_An[0:39] || CSRP_An[Limit]) || 0xFFF.

8.2.3 Accelerator Utilization Record Pointer Registers

8.2.3.1 Accelerator Utilization Record Pointer Zero Register (AURP0_An)

The Accelerator Utilization Record Pointer Zero Register (AURP0_An) contains the upper virtual address, in main storage, of a 64-bit process utilization record (AUR). The PSL increments the AUR value and atomically updates the memory location at the completion of a context time interval.

This register is initialized by the operating system or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

This facility is optional. System software can detect if this feature is supported by writing x‘FF80000000003FFF’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

The value written to the Accelerator Utilization Record Pointer Register is implementation specific.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0010’, where n is an AFU number.

B	KS	KP	N	L	C	TA	LP	Reserved
0 1	2	3	4	5	6	7	8 9	10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Reserved	AURVA_U
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49	50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:1	B	Segment size selector. The segment size selector defines the size of the segment containing the accelerator utilization record.
2	KS	Supervisor (privileged) state storage key (tags inactive). For more information about the KS bit, see the SLB entry section in the *Power ISA, Book III*.
3	KP	Problem state storage key. For more information about the KP bit, see the SLB entry section in the *Power ISA, Book III*.
4	N	No-execute segment if N = ‘1’. For more information about the N bit, see the SLB entry section in the *Power ISA, Book III*.
5	L	Virtual page size selector bit 0. For more information about the L bit, see the SLB entry section in the *Power ISA, Book III*.
6	C	Class. For more information about the C bit, see the SLB entry section in the *Power ISA, Book III*.

Bits	Field Name	Description
7	Reserved	Reserved.
8:9	LP	Virtual page size selector bits 1:2. For more information about the LP field, see the SLB Entry section in the *Power ISA, Book III*.
10:49	Reserved	Set to zeros.
50:63	AURVA_U	Upper bits of the 78-bit virtual address pointer to the accelerator utilization record for the corresponding AFU.

8.2.3.2 Accelerator Utilization Record Pointer One Register (AURP1_An)

The Accelerator Utilization Record Pointer One Register (AURP1_An) contains the lower virtual address, in main storage, of a 64-bit process utilization record (AUR). The AUR value is an estimate of the amount of time a process has used an accelerator function. The AUR value is atomically adjusted by the PSL at the completion of a context time interval or when a process element is completed. The adjustment amount is implementation dependent.

This register is initialized by the operating system or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

This facility is optional. System software can detect if this feature is supported by writing x‘FFFFFFFFFFFFFF80’ to this register and reading back the contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported.

The value written to the Accelerator Utilization Record Pointer One Register (AURP1_An) is implementation specific.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0018’, where n is an AFU number.

AURVA_L
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

AURVA_L	Reserved	V
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56	57 58 59 60 61 62	63

Bits	Field Name	Description
0:56	AURVA_L	Lower bits of the 78-bit virtual address pointer to the accelerator utilization record for the corresponding AFU. **Note:** The lower 7-bits of the 78-bit virtual address pointer are always ‘0’ (that is, 128-byte aligned).
57:62	Reserved	Set to zeros.

Bits	Field Name	Description
63	V	When set, indicates that the accelerator utilization record pointer's virtual address is valid. If the AURP is valid, the PSL updates the AUR in system memory when operating in a shared or AFU-directed programming model.

8.2.4 Storage Segment Table Registers

The following sections define the virtual address pointer to the storage segment table located in system memory. The PSL accesses the associated storage using OS privileges and translation enabled ([HV || PR] = ‘00’ and R = ‘1’) regardless of the PSL_SDR_An settings. The storage containing the segment table must be located in contiguous system memory.

8.2.4.1 Storage Segment Table Pointer Zero Register (SSTP0_An)

The Storage Segment Table Pointer Zero Register (SSTP0_An) contains the upper bits of the storage segment table (SST) virtual address. There is an independent SSTP0_An Register for each AFU. When the AFU is virtualized across partitions, this register is automatically loaded by the PSL from the contents of the process element. Privilege software must set the virtual address using MMIO when the AFU is owned by a single process (that is, the dedicated-process programming model).

This register is initialized by the operating system or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation. System software must perform the following sequence in order, to update the segment table pointer:

1. Disable the segment table pointer by setting the enable bit to ‘0’ in the Segment Table Pointer One Register.
2. Invalidate all the SLB entries by writing the SLB Invalidate All Register (SLBIA_An) for the respective AFU function.
3. Set the new contents for the upper bits of the segment table virtual address pointer in SSTP0_An.
4. Set the new contents for the lower bits of the segment table virtual address pointer in SSTP1_An.

Note: Some implementations can support a cache of effective-to-real-address translations (ERATs) to improve performance. Setting the valid bit to ‘0’ does not invalidate any cached translations. The SLB Invalidate All Register (SLBIA_An) must be used for this purpose.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0020’, where n is an AFU number.

B		KS	KP	N	L	C	Reserved	LP		Reserved	SegTableSize
0	1	2	3	4	5	6	7	8	9	10 11 12 13 14 15 16 17 18 19	20 21 22 23 24 25 26 27 28 29 30 31

Reserved	STVA_U
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49	50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:1	B	Segment size selector. The segment size selector defines the size of the segment containing the segment table.
2	KS	Supervisor (privileged) state storage key. For more information about the KS bit, see the SLB entry section in the *Power ISA, Book III*.
3	KP	Problem state storage key. For more information about the KP bit, see the SLB entry section in the *Power ISA, Book III*.
4	N	No-execute segment if N = ‘1’. For more information about the N bit, see the SLB entry section in the *Power ISA, Book III*.
5	L	Virtual page size selector bit 0. For more information about the L bit, see the SLB entry section in the *Power ISA, Book III*.
6	C	Class. For more information about the C bit, see the SLB entry section in the *Power ISA, Book III*.
7	Reserved	Reserved.
8:9	LP	Virtual page size selector bits 1:2. For more information about the LP field, see the SLB entry section in the *Power ISA, Book III*.
10:19	Reserved	Set to zeros.
20:31	SegTableSize	Segment table size. The segment table size field is used to specify the size of the segment table. This field is used as a mask for the real address calculation for the primary and secondary hash of the segment table. For more information, see *Section 7.1 Storage Segment Table* on page 69. 000000000000 Segment table contains 2 segment table groups (256 bytes). 000000000001 Segment table contains 4 segment table groups (512 bytes). ... 000000001111 Segment table contains 32 segment table groups (4 KB). ... 111111111111 Segment table contains 8K segment table groups (1 MB).
32:49	Reserved	Set to zeros.
50:63	STVA_U	Upper bits of the 78-bit virtual address pointer to the storage segment table for the corresponding AFU.

8.2.4.2 Storage Segment Table Pointer One Register (SSTP1_An)

The Storage Segment Table Pointer One Register (SSTP1_An) contains the lower bits of the storage segment table (SST) virtual address. There is an independent SSTP1_An Register for each AFU. When the AFU is virtualized across partitions, this register is automatically loaded by the PSL from the contents of the process element. Privilege software must set the virtual address using MMIO when the AFU is owned by a single process (that is, the dedicated-process programming model).

This register is initialized by the operating system or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation. System software must perform the following sequence, in order, to update the segment table pointer:

1. Disable the segment table pointer by setting the enable bit to ‘0’ in the segment table pointer one register.
2. Invalidate all the SLB entries by writing the SLB Invalidate All Register (SLBIA_An) for the respective AFU function.
3. Set the new contents for the upper bits of the segment table virtual address pointer in SSTP0_An.
4. Set the new contents for the lower bits of the segment table virtual address pointer in SSTP1_An.

Note: Some implementations can support a cache of ERATs to improve performance. Setting the valid bit to ‘0’ does not invalidate any cached translations. The SLB Invalidate All Register (SLBIA_An) must be used for this purpose.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0028’, where n is an AFU number.

STVA_L
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

STVA_L	Reserved	V
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55	56 57 58 59 60 61 62	63

Bits	Field Name	Description
0:55	STVA_L	Lower bits of the 78-bit virtual address pointer to the storage segment table for the corresponding AFU. **Note:** The lower 8-bits of the 78-bit storage segment table virtual address are always ‘0’ (that is, 256 byte aligned).
56:62	Reserved	Set to zeros. The data for this register is reserved for future use. Writing to this register causes the contents of the segment lookaside buffer to be voided.
63	V	When set, indicates that the storage segment table virtual address is valid.

8.2.5 PSL Authority Mask Register (PSL_AMR_An)

The Virtual Page Class Key Protection mechanism described in the *Power ISA, Book III* provides a means to assign pages of storage to one of 32 classes. The PSL Authority Mask Register (PSL_AMR_An) allows software to modify the access permissions for all accelerator function unit accesses. The PSL_AMR_An is similar to the AMR described the *Power ISA, Book III*.

The access mask for each class defines the access permissions that apply to all read and write operations from the corresponding AFU. The key field value in the page table entry (PTE) corresponds to the class number for each translated virtual address. The key field value in the PTE is used to select the corresponding key in the AMR for the class of the page of storage. For more information about the LPAR, see *Power ISA, Book III*.

Privilege software must initialize the contents of the PSL_AMR_An for the dedicated-process programming model. For the shared or AFU-directed programming model, the content of this register is automatically updated by the PSL using the information in the process element. The new value of the PSL_AMR_An is the process element authority mask (PEAM) value in the process element, masked with the value in the PSL Authority Mask Override Register (PSL_AMOR_An). The store data is masked with the PSL_AMOR_An. The following equations define the value loaded into the PSL_AMR_An Register:

Stores to the privileged 1 memory map address:

PSL_AMOR_An = *store_data*; where *store_data* is the data written using a processor store instruction.

Stores to the privileged 2 memory map address:

PSL_AMR_An = (PSL_AMR_An & ~PSL_AMOR_An) | (*store_data* & PSL_AMOR_An);
where *store_data* is the data written using a processor store instruction.

PSL update using the AMR value in the process element:

PSL_AMR* = (PSL_AMR_An & ~PSL_AMOR_An) | (*PEAM* & PSL_AMOR_An);
where *PEAM* is the data read from the process element by the PSL.

* The PSL_AMR value is used for translation but is not placed into the PSL_AMR_An Register.

This register is initialized by the operating system or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = ‘1’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0030’; where n is an AFU number.

Key_0	Key_1	Key_2	Key_3	Key_4	Key_5	Key_6	Key_7	Key_8	Key_9	Key_{10}	Key_{11}	Key_{12}	Key_{13}	Key_{14}	Key_{15}
0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15	16 17	18 19	20 21	22 23	24 25	26 27	28 29	30 31

Key_{16}	Key_{17}	Key_{18}	Key_{19}	Key_{20}	Key_{21}	Key_{22}	Key_{23}	Key_{24}	Key_{25}	Key_{26}	Key_{27}	Key_{28}	Key_{29}	Key_{30}	Key_{31}
32 33	34 35	36 37	38 39	40 41	42 43	44 45	46 47	48 49	50 51	52 53	54 55	56 57	58 59	60 61	62 63

Bits	Field Name	Description
2*x	$Key_x[0]$	Bit 0 of the access mask for class number x; where $0 \le x \le 31$. Write permissions for AFU writes (**afu_wr**) to virtual pages that contain a key of class x. 0 Writes are permitted. 1 Writes are not permitted.
(2*x) + 1	$Key_x[1]$	Bit 1 of the access mask for class number x; where $0 \le x \le 31$. Read permissions for AFU reads (**afu_rd**) from virtual pages that contain a key of class x. 0 Reads are permitted. 1 Reads are not permitted.

8.2.6 Segment Lookaside Buffer Management Registers

8.2.6.1 SLB Invalidate Entry Register (SLBIE_An)

A write to the SLB Invalidate Entry Register (SLBIE_An) causes the valid (V) bit in all entries of the SLB that match the ESID, class, and segment size to be set to '0'. The remaining fields of each entry are undefined.

This facility is used by the system software to invalidate any noncoherent caches of the segment table or translations using the segment entries.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Address Read/Write

Base Address Offset (P2_Base | P2(n)) + x'0040', where n is an AFU number.

Register Write Field Description:

ESID
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

ESID	C	SS	Reserved	IQ
32 33 34 35	36	37 38	39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61	62 63

Bits	Field Name	Description
0:35	ESID	Effective segment ID.
36	C	Class. The class field is used in conjunction with the SLB Invalidate Entry Register (SLBIE_An). It is used as an additional qualifier for the ESID when multiple virtual address spaces exist.
37:38	SS	Segment size. The segment size field is used in conjunction with the SLB Invalidate Entry Register (SLBIE_An). It is used as an additional qualifier for the ESID when multiple virtual address spaces exist.
39:61	Reserved	Set to zeros.
62:63	IQ	SLB invalidation qualifier (IQ). The IQ field is used to selectively invalidate only the SLB entries based on the ESID, or ESID and PID combination. The value for the PID is defined by the SLBI_Select_An Register. 00 Invalidate the SLBs matching the ESID. 01 Reserved. 11 Invalidate the SLBs matching the ESID and PID.

Register Read Field Description:

Reserved	Max_SLBIEs
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31

Reserved	SLBIEs_Pending
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55	56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:23	Reserved	Set to zeros.
24:31	Max_SLBIEs	Maximum number of SLBIE commands supported. This field indicates the maximum number of outstanding SLB Invalidate Entry commands supported. The value of this field is implementation-dependent.
32:55	Reserved	Set to zeros.
56:63	SLBIEs_Pending	Number of SLBIE commands pending. The SLBIEs_Pending field indicates the number of SLBIE commands currently outstanding. This field is used to determine when the previously issued SLB invalidations are complete. Issuing any additional SLB invalidates (that is, writing this register) when the number of invalidation pending is at the maximum, might result in an SLB invalidate being lost or discarded. x‘00’ No SLB invalidate entry commands are pending. x‘01’ One SLB invalidate entry command is pending. ... x‘FF’ 255 SLB invalidate entry commands are still pending.

8.2.6.2 SLB Invalidate All Register (SLBIA_An)

A write to the SLB Invalidate All Register (SLBIA_An) causes the valid (V) bit in all entries of the SLB to be set to ‘0’, making the entries invalid. The remaining fields of each entry are undefined.

This facility is used by the system software to invalidate any noncoherent caches of the segment table or translations using the segment entries.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0048’, where n is an AFU number.

Register Write Field Description:

Reserved
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Reserved	IH	IQ
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58	59 60 61	62 63

Bits	Field Name	Description
0:58	Reserved	Set to zeros.
59:61	IH	SLB invalidation hint (IH). The IH field is provided by system software as a hint that can be used to selectively invalidate entries in the implementation-specific look aside information. 000 Invalidate all implementation-specific lookaside information. (This is not a hint.) 001 Preserve implementation-specific lookaside information having a class value of ‘0’. 010 Preserve implementation-specific lookaside information created when MSR[IR/DR] = ‘0’. (This encoding is not valid for PSL implementations. Specific to the POWER architecture.) 110 Preserve implementation-specific lookaside information created when MSR[HV] = ‘1’, MSR[PR] = ‘0’, and MSR[IR/DR] = ‘0’. (This encoding is not valid for PSL implementations. Specific to the POWER architecture.) Other All other IH values are reserved.
62:63	IQ	SLB invalidation qualifier (IQ). The IQ field is used to selectively invalidate only the SLB entries based on the PID. The value for the PID is defined by the SLBI_Select_An Register. 00 Invalidate all SLBs. 01 Reserved. 11 Invalidate the SLBs matching the PID.

Register Read Field Description:

Reserved																															1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Reserved																															P
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:30	Reserved	Set to zeros.
31	Reserved	Set to one. This field indicates the maximum number of outstanding SLB Invalidate All commands supported. This field is set to one for TLB Invalidate All commands.
32:62	Reserved	Set to zeros.
63	P	SLB invalidations pending. The SLB invalidation pending (P) field is used to determine when the previously issued SLB invalidations are complete. Issuing any additional SLB invalidates (that is, writing this register) when the number of invalidation pendings is at the maximum, might result in an SLB invalidate being lost or disguarded. 0 No SLB invalidate all commands are pending. 1 An SLB invalidate all command is pending.

Implementation Note:

The SLB Invalidate All Register (SLBIA_An) for a PSL MMU clears the V bit of SLB entry 0. This differs from the Power ISA **slbia** instruction, which does not clear the V bit of SLB entry 0. PSL implementations can only implement the lower bit of the IH field. The PSL does not recognize hints that involve when the implementation-specific lookaside information is created based on the MSR value.

8.2.6.3 SLB Invalidate Selection Register (SLBI_Select_An)

The SLB Invalidate Selection Register (SLBI_Select_An) contains the process identifier (PID) used to select which SLB entries to invalidate.

This facility is used by the system software to specify the PID for a more selective SLB invalidation.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0050’, where n is an AFU number.

PID

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Reserved

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:31	PID	Process identifier (PID). The PID field is used to selectively invalidate only the SLB entries for the specified process ID that match the ESID field.
32:63	Reserved	Reserved.

8.2.7 PSL Data Storage Interrupt Status Register (PSL_DSISR_An)

The PSL Data Storage Interrupt Status Register (PSL_DSISR_An) contains the status that defines the cause of the PSL data storage interrupt. The function of this register is similar to the POWER Data Storage Interrupt Status Register (DSISR). For more information, see the *Power ISA, Book III*.

The content of this register is set by the PSL when a data segment fault (segment table miss) occurs, data storage fault (page table miss) occurs, or a PSL error is detected. The register is reset to zero when the current outstanding fault is either restarted, continued, indicated as an address error, or acknowledged by a write to the PSL Translation Fault Control Register (PSL_TFC_An).

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read Only

Base Address Offset (P2_Base | P2(n))+ x'0060'; where n is an AFU number.

DS	DM	ST	UR	PE	AE	OC	Reserved
0	1	2	3	4	5	6	7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Reserved	M	Reserved	P	A	S	Reserved	K	Reserved
32	33	34 35	36	37	38	39 40 41	42	43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0	DS	Set to '1' if the segment is not found. Bits 32:63 of this register are undefined if this bit is set. This bit is reset to '0' when the corresponding PSL_TFC_An Register is written.
1	DM	Set to '1' if the PTE is not found (same as the M bit) or for a protection fault. Bits 32:63 of this register are defined as shown below. This bit is reset to '0' when the corresponding PSL_TFC_An Register is written.
2	ST	Set to '1' if the PTE is not found for the segment table pointer (SSTP0_An || SSTP1_An). This bit is reset to '0' when the corresponding PSL_TFC_An Register is written.
3	UR	Set to '1' if the PTE is not found for the accelerator utilization record pointer (AURP0_An || AURP1_An). This bit is reset to '0' when the corresponding PSL_TFC_An Register is written.
4	PE	Set to '1' if the PSL has detected an error for the corresponding slice. The reason for the error is implementation specific. This bit is reset to '0' when the corresponding PSL_TFC_An Register is written. **Note:** This is a nontranslation fault interrupt.
5	AE	Set to '1' if the AFU has detected an error and is not able to post the error directly to the associated application. The reason for the error is indicated in the AFU Error Register (AFU_ERR_An). This bit is reset to '0' when the corresponding PSL_TFC_An Register is written. **Note:** This is a nontranslation fault interrupt.
6	OC	Set to '1' if the operating system context warning interval expires. This bit is reset to '0' when the corresponding PSL_TFC_An Register is written. **Note:** This is a nontranslation fault interrupt.
7:32	Reserved	Set to zeros.

Bits	Field Name	Description
33	M	Set to ‘1’ if the PTE is not found. This bit is reset to ‘0’ when the corresponding PSL_TFC_An Register is written.
34:35	Reserved	Set to zeros.
36	P	Set to ‘1’ if the access is not permitted by the storage protection mechanism. This bit is reset to ‘0’ when the corresponding PSL_TFC_An Register is written.
37	A	Set to ‘1’ if an AFU lock type access is to page marked write through or caching inhibited. This bit is reset to ‘0’ when the corresponding PSL_TFC_An Register is written.
38	S	Set to ‘1’ if the access was an **afu_wr** or an **afu_zero** operation. This bit is reset to ‘0’ when the corresponding PSL_TFC_An Register is written.
39:41	Reserved	Set to zeros.
42	K	Set to ‘1’ if the access is not permitted by the virtual-page class key protection. This bit is reset to ‘0’ when the corresponding PSL_TFC_An Register is written.
43:63	Reserved	Set to zeros.

8.2.8 PSL Data Address Register (PSL_DAR_An)

The PSL Data Address Register (PSL_DAR_An) contains the effective address associated with a PSL data segment interrupt or a PSL data storage interrupt. The function of this register is similar to the POWER Data Address Register (DAR). For more information about the DAR, see the *Power ISA, Book III*.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read Only

Base Address Offset (P2_Base | P2(n)) + x‘0068’; where n is an AFU number.

Effective Address

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Effective Address

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:63	Effective Address	Effective address associated with the data segment or data storage interrupt.

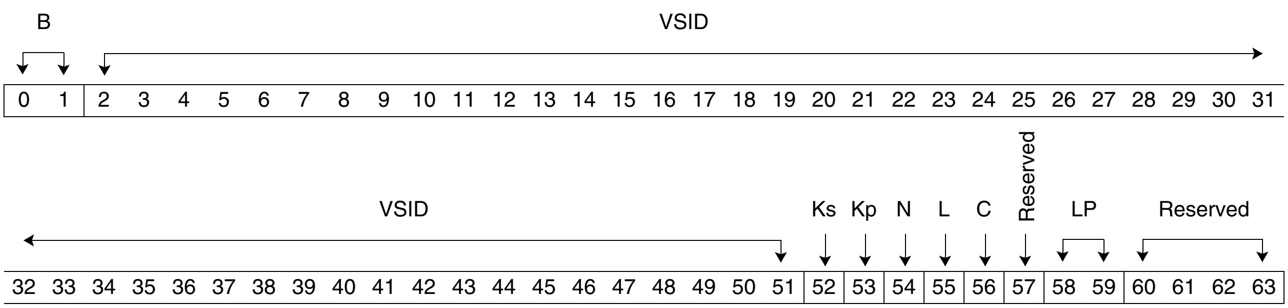
8.2.9 PSL Data Segment Register (PSL_DSR_An)

The PSL Data Segment Register (PSL_DSR_An) contains the segment table entry associated with a PSL data storage interrupt.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read Only

Base Address Offset (P2_Base | P2(n)) + x‘0070’; where n is an AFU number.



Bits	Field Name	Description
0:1	B	Segment size selector. The segment size selector defines the size of the segment for the current data storage fault. For more information about the Ks bit, see the SLB entry section in the *Power ISA, Book III*.
2:51	VSID	Virtual segment ID. The field contains the virtual segment ID for the current data storage fault. For more information about the Ks bit, see the SLB entry section in the *Power ISA, Book III*.
52	Ks	Supervisor (privileged) state storage key. For more information about the Ks bit, see the SLB entry section in the *Power ISA, Book III*.
53	Kp	Problem state storage key. For more information about the Kp bit, see the SLB entry section in the *Power ISA, Book III*.
54	N	No-execute segment if N = ‘1’. For more information about the N bit, see the SLB entry section in the *Power ISA, Book III*.
55	L	Virtual page size selector bit 0. For more information about the L bit, see the SLB entry section in the *Power ISA, Book III*.
56	C	Class. For more information about the C bit, see the SLB entry section in the *Power ISA, Book III*.
57	Reserved	Reserved.
58:59	LP	Virtual page size selector bits 1:2. For more information about the LP field, see the SLB entry section in the *Power ISA, Book III*.
60:63	Reserved	Set to zeros.

8.2.10 PSL Translation Fault Control Register (PSL_TFC_An)

The PSL Translation Fault Control Register (PSL_TFC_An) allows system software to govern the operation of a single slice of the PSL.

Setting the Restart (R) bit of this register causes the PSL transaction with a pending translation fault to be reissued. Hardware resets PSL_TFC_An[R] automatically after a pending PSL command is reissued and the corresponding status bit in the PSL_DSISR_An is reset.

Setting the Continue (C) bit of this register causes the PSL to resume normal operation and report that the pending PSL command has been aborted. Hardware resets PSL_TFC_An[C] automatically after a pending PSL resumes and notifies the AFU that the command has been aborted and the corresponding status bit in the PSL_DSISR_An is reset.

Setting the Address Error (AE) bit of this register causes the PSL to resume operation and report an address error for the pending PSL command to the AFU. Hardware resets PSL_TFC_An[AE] automatically after a pending PSL resumes and notifies the AFU of the addressing error and the corresponding status bit in the PSL_DSISR_An is reset.

Setting the Acknowledge (A) bit of this register informs the PSL that system software has acknowledged the nontranslation fault interrupts reported using the PSL_DSISR_An (PSL error, AFU error, and operating-system context warning). Hardware resets the corresponding status bits in the PSL_DSISR_An when the PSL_TFC_An[A] bit is written to a '1'.

The restart operation is only effective if the PSL command queue is in normal queue operational status. Software must set PSL_TFC_An[R] to '1' to resume a PSL command either after one of the faults listed below or after a page protection fault has been indicated.

- A fault is either a data segment interrupt or a data storage interrupt with the Miss (M) bit set in the PSL Data Storage Interrupt Status Register (PSL_DSISR_An) (see page 133).
- A page protection fault is a data storage interrupt with the Protection (P) bit set in the PSL_DSISR.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x'0078'; where n is an AFU number.

Reserved (0:27)	A	C	AE	R
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27	28	29	30	31

Reserved (32:63)
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:27	Reserved	Reserved.

Bits	Field Name	Description
28	A	Acknowledge interrupt. Resets the nontranslation fault status bits in the PSL_DSISR_An. **Note:** This bit is write-only. A read of the PSL_TFC_An always returns ‘0’ for this bit 0 Nontranslation fault interrupts not acknowledged. 1 Write: Acknowledge the nontranslation faults (PSL error, AFU error, and operating-system context warning. Read: Always returns ‘0’.
29	C	Continue. Current translation fault is not resolved and must be retried at a later time. **Note:** This bit must be set to ‘1’ to restart the PSL and indicate that the current PSL transaction that caused a translation fault is not resolved. The current transaction is aborted. This bit is automatically reset by hardware after the PSL has resumed and indicated a command abort to the AFU. 0 No PSL command continue requested. 1 Write: Restart the PSL with the next transaction (current transaction aborted). Read: PSL waiting to continue.
30	AE	Address error on the PSL transaction caused the translation fault. **Note:** This bit must be set to ‘1’ to restart the PSL and indicate an address error occurred on the PSL transaction that caused a translation fault. This bit is automatically reset by hardware after the PSL has resumed and indicted an address error to the AFU. 0 No PSL command address error requested. 1 Write: Restart the PSL with an address error. Read: PSL address error pending.
31	R	Restarts the PSL transaction that caused the translation fault. **Note:** This bit must be set to ‘1’ to restart the PSL transaction that caused a translation fault. This bit is automatically reset by hardware after the PSL command has been resumed. 0 No PSL command restart requested. 1 Write: Restart the PSL command. Read: PSL command reissue pending.
32:63	Reserved	Reserved.

8.2.11 PSL Process Element Handle Register (PSL_PEHandle_An)

The PSL Process Element Handle Register (PSL_PEHandle_An) contains the current process handle and the AFU command tag. The PE handle field is a 16-bit pointer corresponding to a process element within the scheduled processes area that contains the associated process state, which caused the interrupt. The byte address of the process element is:

((PSL_SPAP_An[SPA Real Address] << 12) | (PSL_PEHandle_An[PE_Handle] << 7))

Note: The start of the scheduled processes area must be naturally aligned to the size of the scheduled processes area.

This register is automatically loaded by the PSL with the corresponding process handle of the process interrupting the system. When an interrupt is not pending, the data returned when reading this register is indeterminate.

This facility is optional for CAIA-compliant devices that do not support virtualization. System software can detect if this feature is supported by writing x‘000000000000FFFF’ to the PSL_LLCMD_An Register and reading back the PSL_LLCMD_An Register contents. If a value of zero is returned, this feature is not supported. Any nonzero value indicates that the feature is supported. The PSL_LLCMD_An Register is read because this register is read only.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read Only

Base Address Offset (P2_Base | P2(n)) + x‘0080’; where n is an AFU number.

Reserved	CMD_Tag
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Reserved	PE_Handle
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:15	Reserved	Reserved.
16:31	CMD_Tag	Command tag. The command tag is a 16-bit tag corresponding to the AFU command that caused the interrupt. This field is indeterminate when the associated interrupt is not the result of an AFU operation. This field is implementation dependent and might not be supported by some implementations. See the implementation-specific documentation for more details.
32:47	Reserved	Reserved.
48:63	PE_Handle	Process element handle. The process element handle is the 16-bit pointer to the corresponding process element. The address of the process element is: ((PSL_SPAP_An[SPA Real Address] << 12) | (PSL_PEHandle_An[PE_Handle] << 7)).

8.2.12 PSL Error Status Register (PSL_ErrStat_An)

PSL Error Status Register (PSL_ErrStat_An) contains the error status for interrupts reported by the PSL_DSISR_An[PE] error. This register is set when the PSL detects a recoverable error on the associated slice.

System software should record the errors reported with this interrupt in the system error log. System software should also write the same value back to this register to reset the status.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x'0088'; where n is an AFU number.

Implementation Dependent
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Implementation Dependent
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:63	Implementation Dependent	Reserved for implementation-dependent PSL errors. Bits in this field are set to '1' if the PSL has detected the corresponding error condition. Additional status information about the error might be available in an implementation-specific register. Bits in this field are reset to '0' when this register is written with the corresponding bit set to '1'.

8.2.13 AFU Control Register (AFU_Cntl_An)

The AFU Control Register (AFU_Cntl_An) allows system software to govern the operation of a single slice of the PSL.

Setting the AFU Slice Enable (E) bit to a ‘1’ causes a start command to be issued to the AFU slice. System software must wait for the status of the AFU to be set to running (AFU_Cntl_An[ES] = ‘100’) before issuing any MMIO to the AFU in a dedicated-process programming model. Software should implement a timeout for enabling an AFU. If the timeout expires and the ES and RS fields are ‘000’ and ‘00’ respectively, then an error was detected by the PSL while enabling the AFU. An error can also be detected if the ES field transitions from a nonzero value to zero. The timeout value should be larger than the PSL implementation-dependent timeout.

Note: The status fields can transition to other states. Software must wait for the AFU Slice Reset Status to be ‘10’ before setting the AFU Slice Enable (E) to a ‘1’.

Setting the AFU Slice Reset (RA) to a ‘1’ causes a reset command to be issued to the AFU slice. System software must wait for the status of the reset to be complete (AFU_Cntl_An[RS] = ‘10’) before enabling the AFU. The reset sequence also causes the AFU to be disabled. The final states of the AFU Slice Enable Status (ES) and the AFU Slice Reset Status (RS) fields are ‘000’ and ‘10’ respectively, after resetting the AFU. Software should implement a timeout for resetting an AFU. If the timeout expires and the ES and RS fields are ‘000’ and ‘00’ respectively, then an error was detected by the PSL while resetting the AFU. An error can also be detected if the RS field transitions from a nonzero value to zero.The timeout value should be larger than the PSL implementation-dependent timeout.

If an error is detected while enabling or resetting an AFU, the AFU should be either downloaded again (for downloadable AFUs) or the CAIA-compliant adapter reset.

The reset and enable status fields (ES, RS) can change when the slice is operating in a virtualized programming model. As part of a switching context, the PSL performs resets and enables of the AFU slice. See *Section 5 Context Management* on page 61 for more information.

Note: The AFU slice enable and reset bit are write only. The status fields for the corresponding request bits indicate when the operation requested has completed.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base | P2(n)) + x‘0090’; where n is an AFU number.

Field	Bits
ES	0 1 2
E	3
RS	4 5
Reserved	6
RA	7
Reserved	8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Reserved	32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:2	ES	AFU slice enable status. Read only. The AFU slice enable status field contains the status of enabling or disabling the AFU. 000 AFU slice disabled. AFU does not respond to MMIO in the dedicated-process model. 010 AFU slice enabled/disable pending. 011 AFU slice enabled pending. Fetching process state for dedicated process. 100 AFU slice enabled. Reset and start commands are complete.
3	E	AFU slice enable. The AFU slice enable bit controls the start and reset commands to the AFU slice. 0 AFU slice disabled. 1 AFU slice enabled.
4:5	RS	AFU slice reset status. Read only. The AFU slice reset status field contains the status of the reset of the AFU. 00 AFU slice not reset. 01 AFU slice reset pending. 10 AFU slice reset sequence is complete.
6	Reserved	Reserved.
7	RA	AFU slice reset. Write only. 0 AFU slice reset not issued. 1 Issue a reset to the AFU slice.
8:63	Reserved	Reserved.

8.2.14 AFU Error Register (AFU_ERR_An)

The AFU Error Register (AFU_ERR_An) contains the error status from the corresponding AFU associated with the AFU error interrupt. This register is used when the AFU is unable to report errors directly to the associated application. The PSL presents this interrupt to the associated OS or hypervisor on IVTE0 or ErrIVTE_Slice when operating in AFU-directed mode.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read Only

Base Address Offset (P2_Base | P2(n)) + x‘0098’; where n is an AFU number.

AFU Error Status

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

AFU Error Status

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:63	AFU Error Status	AFU error status associated with the AFU error interrupt.

8.2.15 PSL WED Register (PSL_WED_An)

The PSL WED Register (PSL_WED_An) contains the current 64-bit value (WED_word_0 and WED_word_1) that is passed to the AFU when a start command is issued.

This register is initialized by the operating system or from the process element. The process element information is used when the PSL scheduled processes area is enabled (PSL_SPAP_An[V] = ‘1’). If initialized by the operating system, this register must be written before enabling the AFU by setting the PSL_AFU_Cntl_An[E] to a ‘1’ from a disabled state (PSL_AFU_Cntl_An[E] = ‘0’).

When the AFU is operating in a virtualized programming model, the data returned when reading this register is indeterminate. CAIA-compliant devices should return the corresponding process element data when an interrupt is pending for diagnostic purposes.

There is one register for each PSL slice. Access to these registers should be privileged. These registers must be accessed using a single 64-bit store operation.

Access Type Read/Write

Base Address Offset (P2_Base I P2(n)) + x‘00A0’; where n is an AFU number.

WED_word_0

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

WED_word_1

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:31	WED_word_0	Word zero of the Work Element Descriptor.
32:63	WED_word_1	Word one of the Work Element Descriptor.

Part II. Accelerator Function Unit Interface

Section 9 AFU Descriptor Overview describes the interface facilities that are provided by the POWER Service Layer (PSL) for the accelerator function units (AFUs). The facilities in these sections of the document provide the AFUs with the ability to read and write main storage, maintain coherency with the system caches, and perform synchronization primitives. Collectively, these facilities are called the accelerator unit interface (AUI).

9. AFU Descriptor Overview

A CAIA-compliant device can support programmable AFUs. The AFU descriptor is a set of registers within the problem state area that contains information about the capabilities of the AFU that is required by system software. The AFU descriptor also contains a standard format for reporting errors to system software. All AFUs must implement an AFU descriptor.

9.1 AFU Descriptor Format

The length of the AFU descriptor is implementation specific. All accesses to the AFU descriptor, including the AFU configuration record, must be either 32-bit or 64-bit operations. The AFU descriptor starts at the offset defined in the vendor-specific, extended configuration record from the start of the privileged 2 area.

The AFU descriptor provides system software with information specific to the AFU. The AFU descriptor also provides a mechanism for assigning regions of the problem state area to system processes attached to the AFU. The assignment is based on the process handle or on the offset of the process element in the linked list. The region assigned to a process handle of x‘0’ corresponds to the beginning of the AFU per-process problem state area (that is, the area starting at the AFU_PSA_offset within the problem state area). The region assigned to a process handle of “n” corresponds to the problem state area starting at AFU_PSA_offset + (n $\times$ AFU_PSA_length $\times$ 4096); where $0 \leq n \leq$ (num_of_processes - 1).

The AFU descriptor contains AFU configuration records that provide system software with the information that is typically provided by the PCIe configuration space if the AFU was a PCIe device. The format of the AFU configuration record can either be the standard 256-byte configuration space or the extended 4 KB configuration space defined by the PCIe specification. If multiple AFU configuration records exist, each record corresponds to a physical function of the AFU. The AFU configuration record space is defined in little-endian format to conform to the PCIe standard.

Note: The length of each configuration record is selectable in 256-byte blocks. The AFU does not have to reserve a full 4 KB for the extended configuration space.

The AFU descriptor also contains an AFU error buffer. The AFU error buffer is intended to be used by the AFU to report application-specific errors. This data can be collected by system software and combined with adapter error data for use in creating error logs or other problem determination.

Note: Some operating systems have a base page size of 64 KB. To be compatible with a base page size of 64 KB, the AFU_PSA_offset should start on a 64 KB boundary and the AFU_PSA_length should also be a multiple of 64 KB. For implementation requirements on the alignment and size of the problem state area, refer to the design guides for the target operating system.

Table 9-1 on page 148 defines the format of the AFU descriptor for a CAIA-compliant device.

Table 9-1. AFU Descriptor (Page 1 of 3)

Register Offset	Field Name	Bits	Description
x'0'	num_ints_per_process	0:15	The power-on reset value of this field specifies the minimum number of interrupts required by the AFU for each process supported. This field is read-only. **Implementation Note:** This value does not include LISN0 used by the PSL for reporting translation faults. A value of zero in this field implies that the AFU does not require any interrupts.
	num_of_processes	16:31	This field specifies the maximum number of processes that can be supported by the AFU. This field can be written by system software to a number less than the power-on value. System software is required to read back the value to determine if the devices support reducing the number of processes supported. **Implementation Note:** If the value written to this field by system software is less than the minimum number of processes required to be supported, an implementation can return the minimum number of processes or the power-on value. For a dedicated process, this field must be set to x'0001'.
	num_of_afu_CRs	32:47	This field specifies the number of configuration records contained in the configuration record area. A length of x'0' indicates that an AFU configuration record does not exist. This is a read-only field.
	req_prog_model	48:63	This field specifies the programming model supported by the AFU. This is a read-only field. Bit Description 48:58 Reserved (set to '0'). 59 Dedicated process (read-only). 0 The AFU does not support dedicated processes. 1 The AFU does support dedicated processes. 60 Reserved (set to '0'). 61 AFU-directed support (read-only). 0 The AFU does not support AFU-directed virtualization. 1 The AFU does support AFU-directed virtualization. 62 Reserved (set to '0'). 63 Shared, time-sliced support (read-only). 0 The AFU does not support shared, time-sliced virtualization. 1 The AFU does support shared, time-sliced virtualization.
x'8' - x'18'	Reserved	0:63	Reserved (set to x'0').
x'20'	Reserved	0:7	Reserved (set to x'0').
	AFU_CR_len	8:63	This field specifies the length of each AFU configuration record in multiples of 256 bytes. If more than one configuration record is present, the total length of the configuration record area is (num_of_CRs $\times$ AFU_CR_len $\times$ 256). A length of x'0' indicates that an AFU configuration record does not exist. This is a read-only field.
x'28'	AFU_CR_offset	0:63	This field specifies the 256-byte aligned offset of the AFU configuration record from the start of the AFU descriptor. This field contains a 64-bit pointer to the start of the AFU configuration records. The lower 8 bits of the pointer are always '0' (256 byte aligned). This is a read-only field.

Table 9-1. AFU Descriptor (Page 2 of 3)

Register Offset	Field Name	Bits	Description
x'30'	PerProcessPSA_control	0:7	Bit / Description 0:5 Reserved (set to '0'). 6 Per-process problem state area required (read-only). 0 A per-process problem state area is not required. 1 A per-process problem state area is required. The per-process problem state area is a subset of the overall problem state area. The problem state area required bit must also be set if this bit is set. 7 Problem state area required (read-only). 0 A problem state area is not required. Only the necessary area for the AFU descriptor, configuration records, and error buffers area are mapped into the system address space. 1 A problem state area is required.
	PerProcessPSA_length	8:63	If the per-process problem state area required bit is set, this field specifies the length of each per-process problem state area, in multiples of 4 KB. The size of per-process problem state area required is determined by: PerProcess_area = PerProcessPSA_length × 4K × num_of_processes If the per-process problem state area required bit is not set, this field is reserved and returns x'0'. This is a read-only field. **Implementation Note:** Operating systems using a base page size of 64 KB might require the problem state area to be a multiple of 64 KB. To assign different regions of the problem state area to each process (PerProcessPSA_control[6] = '1'), each region might be required to be a multiple of 64 KB. See the target operating system details for more information.
x'38'	PerProcessPSA_offset	0:63	This field specifies the 4 KB aligned offset of the per-process problem state area from the start of the problem state area. This field contains a 64-bit pointer to the start of the per-process problem state area. The lower 12-bits of the pointer are always '0' (4 KB aligned). This is a read-only field. **Implementation Note:** Operating systems using a base page size of 64 KB might require the problem state area to be aligned on a 64 KB boundary. To assign different regions of the problem state area to each process (PerProcessPSA_control[6] = '1'), each region might be required to be aligned on a 64 KB boundary. See the target operating system details for more information.
x'40'	Reserved	0:7	Reserved (set to x'0').
	AFU_EB_len	8:63	This field specifies the length of the AFU error buffer in multiples of 4 KB. A length of x'0' indicates that an AFU error buffer does not exist. This is a read-only field.

Table 9-1. AFU Descriptor (Page 3 of 3)

Register Offset	Field Name	Bits	Description
x'48'	AFU_EB_offset	0:63	This field specifies the 4 KB aligned offset of the AFU error buffer information from the start of the AFU descriptor. This field contains a 64-bit pointer to the start of the AFU error status information. The lower 12-bits of the pointer are always '0'. This is a read-only field.
x'50'	Paged Resolution Handle	0:63	Bit / Description 0 — The AFU supports receiving paged resolution responses from the OS (read only) 0 The AFU ignores writes to x'50' and x'58'. 1 The AFU requires the OS to write x'50' and x'58' for the resolution of any data storage/segment interrupt where the OS responded with Continue (PSL_TFC_An[C] = '1'). 1:15 — Reserved. 16:31 — Command Tag. The tag of the operation being resolved by the resolution MMIO sequence. The value written to this field is the value provided by the PSL_PEHandle_An[CMD_Tag]. **Implementation Note:** This field is implementation dependent and might not be supported by some implementations. See the implementation-specific documentation for more details. 32:47 — Reserved. 48:63 — Process Handle of the paged transaction that has been resolved. This is a read/write field.
x'58'	Paged Resolution EA	0:63	Bit / Description 0:51 — Effective address of the paged transaction that has been resolved. 52:55 — Resolution. x'0' Page fault resolved x'1' Addressing error x'2' Protection fault on a read operation x'3' Protection fault on a write operation 56:57 — Reserved. 58:63 — Page Size Mask (the number of bits to ignore for EA compare). x'0' 4 KB page x'1' 8 KB page x'2' 16 KB page ... and so on. This is a write only field.

9.1.1 Paged-Response Resolution

When system software cannot immediately resolve a fault generated by the PSL, a Continue response is written to the PSL_TFC_An Register (PSL_TFC_An[C] = '1'). The corresponding AFU commands also receive a Paged response indicating that the requested operation will not be completed. Offset x'50' and x'58' in the AFU descriptor provide a mechanism for the AFU to be notified when the fault has been resolved by system software and the how the fault was resolved.

When the AFU supports receiving a paged-resolution response (AFU descriptor offset x'50' bit 0 = '1'), system software must issue a write to the AFU descriptor at offset x'50' followed by a write to offset x'58' for every operation that results in a data storage/segment interrupt (DSI), where the response is a Continue response (PSL_TFC_An[C] = '1'). This MMIO sequence is referred to as the resolution MMIO sequence unless the process has been terminated. AFU descriptor offset x'50' is written with the value read from the PSL_PEHandle_An for the corresponding DSI.

System software does not guarantee any upper bound on the time between the Continue response and the resolution MMIO sequence. Furthermore, the resolution MMIO sequence is not guaranteed to be in the same order as the Continue responses.

System software must guarantee that after a process has been terminated, all the resolution MMIO sequences will be performed or discarded before the process element is removed. After the process element remove command has been issued, there shall not be any resolution MMIO sequences issued for that process.

An AFU that blocks waiting for the MMIO sequence to restart an operation should also implement a time-out for the case where the requested data is paged out of system memory. How the command is restarted is implementation specific.

Part III. PCIe Configuration Environment

This chapter describes the PCIe configuration space for a CAIA-compliant device. The configuration facilities described in these sections are compatible with the PCIe Generation 3 architecture and allow system software to set the base address, the message signaled interrupt (MSI-X) capabilities, and various other configuration parameters for the CAIA-compliant device. In addition, the following sections describe the procedure to verify that the PCIe device is CAIA compliant and is signed by an authorized vendor. Collectively, these facilities are called the PCIe Configuration Environment (PCE).

10. PCIe Configuration Overview

The PCIe configuration space for a CAIA-compliant device follows the PCIe standard. The configuration registers in this section should only be accessed by system software using the PCIe Configuration Mechanism. See the *I/O Design Architecture v2 (IODA2) (Version 2.4+)* for more information about how to access these registers in a POWER system. The PCIe configuration facilities include the following sections:

- *Section 10.1 PCIe Type 0 Configuration Space* on page 155
- *Section 10.2 PCIe MSI-X Capability* on page 159
- *Section 10.3 CAIA Vendor-Specific Extended Capability Structure* on page 160
- *Section 10.4 Dual Bus Configuration Space* on page 167

The PCE section of the CAIA is not intended to reproduce the PCIe standard for the configuration space but rather to define how the various configuration fields are used by a CAIA-compliant device. See the PCIe standard for more complete information on the PCIe configuration space.

Note: The bit and byte numbering for the registers in the PCIe Configuration Environment is little endian. This is different than all other registers in the CAIA. Little-endian numbering refers to the following:

- The least significant byte is at offset x‘3’ in a 32-bit word.
- The least significant bit is number 0.

10.1 PCIe Type 0 Configuration Space

The Coherent Accelerator Interface Architecture (CAIA) uses a PCIe type 0 configuration space for coherent accelerators connected using a coherent protocol tunneled over a PCI Express bus.

Table 10-1 defines the PCIe type 0 configuration header for a CAIA-compliant device.

Note: The configuration space is defined in little-endian format to conform to the PCIe standard. This diverges from the other facilities in the CAIA, which are defined in a big-endian format.

Note: All the base address registers (BARs) are 64-bit addresses. Each BAR requires two configuration words in the type 0 header to represent a single address space.

Table 10-1. PCIe Type 0 Configuration Header (Page 1 of 4)

Field Name	Configuration Header Offset	Bits	Description
Device ID	x‘0’	31:16	The Device ID is an implementation-specific field. The Device ID field is the Device_ID[31:24] (byte 3) concatenated with the Device_ID[23:16] (byte 2).
Vendor ID	x‘0’	15:0	The IBM Vendor ID is an implementation-specific field. The Vendor ID field is the Vendor_ID[31:24] (byte 3) concatenated with the Vendor_ID[23:16] (byte 2).
Status	x‘4’	31:16	See the PCIe 3.0 Specification.
Command	x‘4’	15:0	See the PCIe 3.0 Specification.
Class Code	x‘8’	31:8	The Class Code is an implementation-specific value for bi-modal devices set to operate in PCIe mode. The Class Code field value is x‘120000’ for all CAPI devices, including bi-modal devices set to CAPI mode. The Class Code is the Class_Code[31:24] (byte 3) concatenated with the SubClass_Code[23:16] (byte 2) concatenated with the Programming_Interface[15:8] (byte 1).
Revision ID	x‘8’	7:0	The Revision ID is an implementation-specific field.
BIST	x‘C’	31:24	The built-in self test (BIST) is an implementation-specific field.
Header Type	x‘C’	23:16	Read-only field. Set to x“00’.
Master latency Timer	x‘C’	15:8	Read-only field. Set to x‘00’.
Cache Line Size	x‘C’	7:0	Read-only field. Set to x‘00’.
Base Address Register 0 (P2_Base_low)	x‘10’	31:0	Base Address Register 0 is an implementation-specific field for bi-modal devices set to operate in PCIe mode. For all CAPI devices, including bi-modal devices set to CAPI mode, Base Address Register 0 is defined below: • Least significant 32-bits of the 64-bit base address for the Privileged 2/problem state register space (P2_Base). • Bits n:0 are read-only and always returned as ‘0’ for reads of the base register; where 2^n is the size of the CAIA register map. • BAR 0/1 must be greater than or equal to 4 GB. • Default value: x‘00000004’. **Note:** Some implementations require the address specified by this BAR to be greater than or equal to 4 GB, so that 64-bit addressing is used when addressing the MMIO region. **Note:** This register is represented as little-endian where the most significant bit is bit 63 and the least significant bit is bit 0. This register is not used for the secondary port of a dual port PCIe CAIA-compliant device.

Table 10-1. PCIe Type 0 Configuration Header (Page 2 of 4)

Field Name	Configuration Header Offset	Bits	Description
Base Address Register 1 (P2_Base_high)	x‘14’	31:0	Base Address Register 1 is an implementation-specific field for bi-modal devices set to operate in PCIe mode. For all CAPI devices, including bi-modal devices set to CAPI mode, Base Address Register 1 is defined below: • Most significant 32-bits of the 64-bit base address for the Privileged 2/problem state register space (P2_Base). • Bits n:0 are read-only and always returned as ‘0’ for reads of the base register; where 2^n is the size of the CAIA problem state register map. • BAR 0/1 must be greater than or equal to 4 GB. • Default value: x‘00000000’. **Note:** Some implementations require the address specified by this BAR to be greater than or equal to 4 GB so that 64-bit addressing is used when addressing the MMIO region. **Note:** This register is represented as little-endian, where the most significant bit is bit 63 and the least significant bit is bit 0. This register is not used for the secondary port of a dual port PCIe CAIA-compliant device.
Base Address Register 2 (P1_Base_low)	x‘18’	31:0	Base Address Register 2 is an implementation-specific field for bi-modal devices set to operate in PCIe mode. For all CAPI devices, including bi-modal devices set to CAPI mode, Base Address Register 2 is defined below: • Least significant 32-bits of the 64-bit base address for the Privileged 1 register space (P1_Base). • Bits n:0 are read-only and always returned as ‘0’ for reads of the base register; where 2^n is the size of the CAIA register map. • Default value: x‘00000004’. **Note:** Some implementations require the address specified by this BAR to be greater than or equal to 4 GB so that 64-bit addressing is used when addressing the MMIO region. **Note:** This register is represented as little endian where the most significant bit is bit 63 and the least significant bit is bit 0. This register is not used for the secondary port of a dual port PCIe CAIA-compliant device.
Base Address Register 3 (P1_Base_high)	x‘1C’	31:0	Base Address Register 3 is an implementation-specific field for bi-modal devices set to operate in PCIe mode. For all CAPI devices, including bi-modal devices set to CAPI mode, the Base Address Register 3 is defined below: • Most significant 32-bits of the 64-bit base address for the Privileged 1 register space (P1_Base). • Bits n:0 are read-only and always returned as ‘0’ for reads of the base register; where 2^n is the size of the CAIA problem state register map. • Default value: x‘00000000’. **Note:** Some implementations require the address specified by this BAR to be greater than or equal to 4 GB, so that 64-bit addressing is used when addressing the MMIO region. **Note:** This register is represented as little endian where the most significant bit is bit 63 and the least significant bit is bit 0. This register is not used for the secondary port of a dual port PCIe CAIA-compliant device.

Table 10-1. PCIe Type 0 Configuration Header (Page 3 of 4)

Field Name	Configuration Header Offset	Bits	Description
Base Address Register 4 (CAPI_Base_low)	x‘20’	31:0	Base Address Register 4 is an implementation-specific field for bi-modal devices set to operate in PCIe mode. For all CAPI devices, including bi-modal devices set to CAPI mode, the Base Address Register 4 is defined below: • Least significant 32-bits of the 64-bit base address for the CAPI protocol area. • Bits 47:0 are read-only and always returned as ‘0’ for reads of the base register. • System software must set BAR 4/5 to the CAPI Protocol address range compatible with the processor implementation. • Default Value: x‘00000004’. This register is represented as little endian where the most significant bit is bit 63 and the least significant bit is bit 0. This register is not used for the secondary port of a dual port PCIe CAIA-compliant device.
Base Address Register 5 (CAPI_Base_high)	x‘24’	31:0	Base Address Register 5 is an implementation-specific field for bi-modal devices set to operate in PCIe mode. For all CAPI devices, including bi-modal devices set to CAPI mode, the Base Address Register 5 is defined below: • Most significant 32-bits of the 64-bit base address for the CAPI protocol area. • System software must set BAR 4/5 to the CAPI protocol address range compatible with the processor implementation. • Default Value: x‘00000000’. This register is represented as little endian where the most significant bit is bit 63 and the least significant bit is bit 0. This register is not used for the secondary port of a a dual port PCIe CAIA-compliant device.
Cardbus CIS Pointer	x‘28’	31:0	Read-only field. Set to x‘00’.
Subsystem ID	x‘2C’	31:16	The Subsystem ID is an implementation-specific field. The Subsystem ID field is the Subsystem_ID[31:24] (byte 3) concatenated with the Subsystem_ID[23:16] (byte 2).
Subsystem Vendor ID	x‘2C’	15:0	The IBM Subsystem Vendor ID is an implementation-specific field. The Subsystem Vendor ID field is the Subsystem_Vendor_ID[31:24] (byte 3) concatenated with the Subsystem_Vendor_ID[23:16] (byte 2).
Expansion ROM Base Address	x‘30’	31:0	See the PCIe 3.0 Specification. This configuration register is optional for a CAIA-compliant device. The contents of the ROM is implementation specific.
Reserved	x‘34’	31:8	Read-only field. Set to x‘00’.
Capabilities Pointer	x‘34’	7:0	The Capabilities Pointer is an implementation-specific field. CAIA-compliant devices should include the vital product data (VPD) capability structure defined by the PCI Local Bus Specification (ID = x‘03’). At a minimum, the VPD information should contain the part number (PN), engineering change level (EC), and serial number (SN) fields.
Reserved	x‘38’	31:0	Read-only field. Set to x‘00’.
Max_Lat	x‘3C’	31:24	Read-only field. Set to x‘00’.

Table 10-1. PCIe Type 0 Configuration Header (Page 4 of 4)

Field Name	Configuration Header Offset	Bits	Description
Min_Gnt	x'3C'	23:16	Read-only field. Set to x'00'.
Interrupt Pin	x'3C'	15:8	Implementation-specific field. This value of this field has no effect when operating in CAPI mode.
Interrupt Line	x'3C'	7:0	Implementation-specific field. This value of this field has no effect when operating in CAPI mode.

10.2 PCIe MSI-X Capability

The MSI-X capability mechanism is used to identify and configure the interrupt structure for a CAIA-compliant accelerator. The CAIA supports two MSI-X modes: single MSI-X table entry and full MSI-X table.

When operating with a single MSI-X table entry, interrupts are not fully compliant with the PCIe architecture. The PCIe architecture requires a MSI-X table and a pending-bit array (PBA) in MMIO space. A CAIA-compliant accelerator does not use the MSI-X table or the PBA. In place of the table, the interrupt message address and data are constructed from the PSL_IVTE_Offset_An and PSL_IVTE_Limit_An Registers. See *Section 6 Interrupts* on page 63 and *Section 8.1.15 PSL IVTE Limit Register (PSL_IVTE_Limit_An)* on page 95 for more information.

When operating with a full MSI-X table, interrupts are compliant with the PCIe architecture. The MSI-X table and a pending bit array in MMIO space are used. The index into the MSI-X table is constructed from the PSL_IVTE_Offset_An and PSL_IVTE_Limit_An Registers. See *Section 6 Interrupts* on page 63 and *Section 8.1.15 PSL IVTE Limit Register (PSL_IVTE_Limit_An)* on page 95 for more information. The address and data presented for the interrupt is provided by the MSI-X table entry selected.

10.2.1 MSI-X Capability Structure

Table 10-2 is the MSI-X capability structure for a CAIA-compliant accelerator.

Table 10-2. CAIA MSI-X Capability Structure

<table>
<tr><th rowspan="2">Index CP +</th><th colspan="32">Configuration Data</th></tr>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td colspan="16">Message Control</td><td colspan="8">Next Capability Pointer</td><td colspan="8">Capability ID</td></tr>
<tr><td>4</td><td colspan="29">Table Offset</td><td colspan="3">BIR</td></tr>
<tr><td>8</td><td colspan="29">PBA Offset</td><td colspan="3">BIR</td></tr>
</table>

Power Service Layer Implementation Detail:

Table Offset || BIR (offset x‘4’) = x‘8002’; MSI-X Table located at offset x‘8000’ from BAR 2 (Privileged 2 area).
PBA Offset || BIR (offset x‘8’) = x‘7F02’; pending bit array located at offset x‘7F00’ from BAR 2 (Privileged 2 area).

Note: The MSI-X table and PBA offsets are set in the MSI-X capability structure and point to a valid MMIO location under the base address selected by the BIR field. The values of the MSI-X table and PBA offsets are implementation dependent.

10.3 CAIA Vendor-Specific Extended Capability Structure

The Vendor-Specific Extended Capability (VSEC) structure defines the unique capabilities of the CAIA-compliant device. The CAIA uses the PCIe capability list architecture to link a VSEC structure within the device's configuration space. The capability list anchor is at offset x'100' in the configuration space defined in *Section 10.1* on page 155.

Table 10-3 defines the format of the VSEC structure for CAIA-compliant devices.

Table 10-3. VSEC Format

<table>
<tr><th rowspan="3">Addr</th><th colspan="32">VSEC Data</th></tr>
<tr><th colspan="8">Byte 3 / Address 3</th><th colspan="8">Byte 2 / Address 2</th><th colspan="8">Byte 1 / Address 1</th><th colspan="8">Byte 0 / Address 0</th></tr>
<tr><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0x0</td><td colspan="12">Next Capability Pointer</td><td colspan="4">Cap_Ver (0x1)</td><td colspan="16">Capability ID (0x000B)</td></tr>
<tr><td>0x4</td><td colspan="12">VSEC Length (0x080)</td><td colspan="4">VSEC_Rev (0x0)</td><td colspan="16">VSEC ID (0x1280)</td></tr>
<tr><td>0x8</td><td colspan="8">Reserved</td><td colspan="8">Mode Control</td><td colspan="8">Status</td><td colspan="8">Number of AFUs</td></tr>
<tr><td>0xC</td><td colspan="16">CAIA Version</td><td colspan="16"><u>PSL</u> Revision Level</td></tr>
<tr><td>0x10</td><td>IL</td><td>R</td><td colspan="2">IC</td><td colspan="12">Reserved</td><td colspan="16">Base Image Revision Level</td></tr>
<tr><td>0x14:
0x1C</td><td colspan="32">Reserved</td></tr>
<tr><td>0x20</td><td colspan="32">AFU Descriptor Offset</td></tr>
<tr><td>0x24</td><td colspan="32">AFU Descriptor Size</td></tr>
<tr><td>0x28</td><td colspan="32">Problem State Offset</td></tr>
<tr><td>0x2C</td><td colspan="32">Problem State Size</td></tr>
<tr><td>0x30:
0x3C</td><td colspan="32">Reserved</td></tr>
<tr><td>0x40</td><td colspan="32">PSL Programming Port (optional)</td></tr>
<tr><td>0x44</td><td colspan="32">PSL Programming Control (optional)</td></tr>
<tr><td>0x48:
0x4C</td><td colspan="32">Reserved</td></tr>
<tr><td>0x50</td><td colspan="32">Flash Address Register (optional)</td></tr>
<tr><td>0x54</td><td colspan="32">Flash Size Register (optional)</td></tr>
<tr><td>0x58</td><td colspan="32">Flash Status / Control Register (optional)</td></tr>
<tr><td>0x5C</td><td colspan="32">Flash Data Port (optional)</td></tr>
<tr><td>0x60:
0x7C</td><td colspan="32">Reserved</td></tr>
</table>

Table 10-4. VSEC Description (Page 1 of 6)

Field Name	VSEC Offset	Bits	Description
Capability ID	x'0'	15:0	This field identifies the type of capability structure. This field is set to x'000B' for a VSEC structure.
Capability Version	x'0'	19:16	The Capability Version field is defined by the PCI Special Interest Group (PCI-SIG). The value for this field should be x'1'.
Next Capability Pointer	x'0'	31:20	Pointer to the next capability structure in the list. A value of x'000' indicates that this VSEC is the last capability structure in the list. A nonzero value is a pointer to the next capability structure in the list.
VSEC ID	x'4'	15:0	This field identifies the type of VSEC structure. This field is set to x'1280' for CAIA-compliant devices.
VSEC Revision	x'4'	19:16	The revision level of the VSEC structure. For this level of the CAIA, the VSEC revision is x'0'.

Table 10-4. VSEC Description (Page 2 of 6)

Field Name	VSEC Offset	Bits	Description
VSEC Length	x‘4’	31:20	The length, in bytes, of the VSEC structure included the header at offset x‘0’. For this level of the CAIA, the VSEC length is x‘080’.
Number of AFUs	x‘8’	7:0	The number of AFUs is a read/write field that defines the number of AFUs supported by the device. System software can write this field to a value less than the power-on value to reduce the number of supported AFUs. After reducing the number of supported AFUs, system software must read back this field to determine if the device supports reducing the number of supported AFUs. For devices with a fixed PSL, this field will power-on to the number of supported AFUs. For devices that support a loadable PSL, this field will power-on with a value of x‘0’, if a PSL is not present at power on; or with the number of supported AFUs if a PSL is present at power on. This field can change, after a PSL is downloaded, to the number of supported AFUs by the downloaded PSL.
Status	x‘8’	15:8	Status. Bits Description 15 Secondary PCIe Links for CAPI (read-only). This bit is used to indicate whether the corresponding PCIe link is an extension port used for more bandwidth. 0 Corresponding PCIe link is a primary PCIe port. 1 Corresponding PCIe link is a secondary PICe port for CAPI bandwidth enhancement. 14:13 MSI-X Address Selection (read-only). This bit is used to indicate the type of MSI-X address supported by the device. 00 Fixed MSI-X address. The ISN calculated by the PSL_IVTE_Offset_An and PSL_IVTE_Limit_An values concatenated with 4 bits of zero (IVTE || 0x0) is ORed with a platform-specific constant (x‘1000000000000000’). 01 Single MSI-X table entry. The ISN calculated by the PSL_IVTE_Offset_An and PSL_IVTE_Limit_An values concatenated with 4 bits of zero (IVTE || 0x0) is ORed with the address provided in the entry of the MSI-X table pointed to by the MSI-X capability structure. 10 Full MSI-X table. The ISN calculated by the PSL_IVTE_Offset_An and PSL_IVTE_Limit_An values is an index into the MSI-X table pointed to by the MSI-X capability structure. 12 Reserved. 11:10 Flash status (read-only). 00 Flash is not present. 01 Flash is present but can only be read. 10 Flash is present and can be programmed. 11 Reserved. 9 Loadable AFUs (read-only). This bit applies to all AFUs. 0 AFUs are not loadable unless the AFU is imbedded in the PSL image. AFU designs are fixed in hardware or part of the PSL image or hardware design. 1 AFUs are loadable. AFU designs can be downloaded by system software. 8 Loadable PSL (read-only). 0 PSL is not loadable. The PSL design is fixed in hardware. 1 PSL is loadable. The PSL design can be downloaded during initialization time.

Table 10-4. VSEC Description (Page 3 of 6)

Field Name	VSEC Offset	Bits	Description
Mode Control	x'8'	23:16	Mode Control Register. Bits Description 23:21 CAPI protocol area size. At power-on or following a PCIe reset, this field indicates the sizes of the CAPI protocol area (BAR 4/5) supported by the CAIA-compliant device. Before enabling the device, system software must set this field to the size of the CAPI protocol area compatible with the host system. System software must set only one of these bits to '1'. The other bits must be set to '0'. 100 1024 TB CAPI protocol area. 010 512 TB CAPI protocol area. 001 256 TB CAPI protocol area. All other values are reserved. **Implementation Note:** System software must set the protocol area size to 256 TB for POWER8 systems. 20:17 Reserved. 16 CAPI Protocol Enable. This bit is used to enable the use of the CAPI protocol for the device. Devices that are only capable of CAPI mode initialize this bit to '1' and treat the bit as read-only. 0 Operate as a PCIe device (PCIe mode). 1 Operate as a CAIA-compliant device (CAPI mode).
Reserved	x'8'	31:24	Reserved. Set to x'00000000'.
CAIA Version	x'C'	31:16	This field identifies the CAIA version that the device implements. This field might not be valid until after the PSL is downloaded for devices that support loadable PSLs. This field is divided into a major and minor number (X.y; where X is the major version number and 'y' is the minor version number). Bits Description 31:24 Major version number ('X'). 23:16 Minor version number ('y').
PSL Revision Level	x'C'	15:0	This field identifies the revision level of the PSL design. This field might not be valid until after the PSL is downloaded for devices that support loadable PSLs.
Image Loaded	x'10'	31	Bits Description 31 Image loaded (read-only). This bit is the status of the image loaded into a programmable CAIA-compliant implementation. When this bit is set to '1' indicating that the factory image was loaded, it typically indicates that the user's image was corrupted. 0 Factory image. 1 User's image.
Reserved	x'10'	30	Reserved. Set to '0'.

Table 10-4. VSEC Description (Page 4 of 6)

Field Name	VSEC Offset	Bits	Description
Image Control	x‘10’	29:28	Bits / Description 29: Request image reload on the next PERST. This bit is used to cause a flash image load for programmable CAIA-compliant implementation. If this bit is set to ‘0’, a power cycle of the adapter is required to load a programmable CAIA-compliant implementation from flash. 0 Assertion of PERST does not cause a flash image load. 1 Assertion of PERST causes a flash image load. 28: Image selection. This bit is used to select which image is loaded when requesting an image reload on the next PERST. This bit has no effect on the power cycle of the adapter. 0 Factory flash image. 1 User flash image.
Reserved	x‘10’	27:16	Reserved. Set to ‘0’.
Base Image Revision Level	x‘10’	15:0	This field identifies the revision level of the base image for devices that support loadable PSLs. For programmable devices such as FPGAs, this field identifies the image contained in the on-adapter flash that is loaded during the initial program load (that is reset or power-on).
Reserved	x‘14’ - x‘1C’	31:0	Reserved. Set to x‘00000000’.
AFU Descriptor Offset	x‘20’	31:0	This field specifies the offset of the AFU descriptor from the start of the privileged 2 BAR. The field contains bits 0:31 of a 48-bit offset. The lower 16-bits of the offset are always ‘0’. This is a read-only field. For CAIA-compliant devices with multiple AFUs, this field points to the AFU descriptor for AFU(0). The AFU descriptor offset for AFU(n) is located at: AFU(n) descriptor = (AFU Descriptor Offset * 64K) + (AFU Descriptor Size * 64K * n)
AFU Descriptor Size	x‘24’	31:0	This field specifies the maximum length of each AFU descriptor. The field is the maximum number of 64 KB blocks reserved for each AFU descriptor. This is a read-only field.
Problem State Offset	x‘28’	31:0	This field specifies the offset of the AFU problem state area from the start of the privileged 2 BAR. The field contains bits 0:31 of a 48-bit offset. The lower 16-bits of the offset are always ‘0’. This is a read-only field. For CAIA-compliant devices with multiple AFUs, this field points to the problem state area for AFU(0). The problem state area offset for AFU(n) is located at: AFU(n) problem state = (Problem State Offset * 64K) + (Problem State Size * 64K * n)
Problem State Size	x‘2C’	31:0	This field specifies the maximum length of each AFU’s problem state area. The field is the maximum number of 64 KB blocks of problem state area reserved by the PSL for each AFU. This is a read-only field.
Reserved	x‘30’ - x‘3C’	31:0	Reserved. Set to x‘00000000’.
PSL Programming Port	x‘40’	31:0	This configuration register is used to load a PSL for devices that support a downloadable PSL. This register can also be used to download AFUs. Writes to this configuration register provide the programming information. The architecture for the programming port is device specific.

Table 10-4. VSEC Description (Page 5 of 6)

Field Name	VSEC Offset	Bits	Description
PSL Programming Control	x'44'	31:0	This configuration register is used to load a PSL for devices that support a downloadable PSL. This register can also be used to download AFUs. Writes to this configuration register provide the programming information. The architecture for the programming port is device specific. The architecture for the first CAPI programmable device follows: Bits Description 15:0 Free space (read-only). The number of words that can be written to the configuration port. Implementations can have a queue for the programming information. This field allows software to poll for the amount of room in the programming information queue. 16 PR ready (read-only). 0 Internal reconfiguration machine is not ready. 1 Internal reconfiguration machine is ready. 17 PR done (read-only). 0 Internal reconfiguration machine is not done. 1 Internal reconfiguration machine is done. 20:18 Programming status. 000 Reset. 001 Programming error. 010 CRC error. 011 Programmnig stream incompatible. 100 Programming in progress. 101 Programming successful. 110 Reserved. 111 Reserved. 30:21 Reserved. Set to '0'. 31 PR request (R/W). This bit is written to a '1' to request the programming of the PSL. This bit is reset to '0' at the completion of the programming sequence or if an error is detected. 0 No programming request. 1 Request to program the PSL.
Flash Address Register	x'50'	31:0	This configuration register contains the starting address within the configuration flash device for program and read operations. The value in this register is a word (that is, 4-byte) address. Support of a configuration flash is optional.

Table 10-4. VSEC Description (Page 6 of 6)

Field Name	VSEC Offset	Bits	Description
Flash Size Register	x'54'	31:0	This configuration register contains the size for flash program and read operations. For program operations, the size is the number of blocks minus 1 where the block size is 64K words (256 KB). For read operations, the size is the number of words (4 bytes) minus 1. Support of a configuration flash is optional.
Flash Status / Control Register	x'58'	31:0	This configuration register contains the status and control for flash program and read operations. Bits Description 31 Flash ready (read-only). 0 Flash interface is not ready. 1 Flash interface is ready. A program or read request can be performed. 30 Operation done (read-only). 0 Current flash program or read operation is not complete. 1 Current fash program or read operation is complete. 27 Read request (write to '1', hardware reset to '0'). 0 No read of flash requested. 1 Flash read request. 26 Program request (write to '1', hardware reset to '0') 0 No program of flash requested. 1 Flash program request. 15 Erase status (read-only) 0 Erase of flash is complete. (Next step is programming.) 1 Erase of flash is in process. 14 Programming status (read-only). This bit is not synchronized with the programming write operations. Software should only use this bit for diagnostics. 0 Programming of flash is complete. 1 Programming of flash is in process. 13 Read status (read-only). Implementations can perform a prefetch of flash. Software must only use this bit for diagnostics and not for a status when all reads by software have been performed. 0 Read of flash is complete. 1 Read of flash is in process. 9:0 Remaining operations (read-only). This field contains the number of block remaining to be erased or programmed for a program request. For a read request, this field reflects the number of outstanding reads of the flash that are remaining. Updates of this field are not synchronized with configuration reads or writes. Software should only use this field for diagnostics. All other bits are reserved and set to '0'. Support of a configuration flash is optional.
Flash Data Port	x'5C'	31:0	This configuration register is used for writing the programming data for a program request and for reading the flash data for a read request.

10.3.1 PSL Programming Procedure

The PSL must only be downloaded following a PCIe reset. The procedure for programming the PSL follows:

1. Set the PR Request bit in the PSL Programming Control field.
2. Wait for the PR Ready bit in the PSL Programming Control field to be set by the device.
3. Read the amount of free space in the PSL Programming Control field and the other status information. If free space is available or an error has occurred, continue with the next step.
4. If an error has occurred (CRC Error or PR Error bits are set to ‘1’), then go to step 7.
5. If PR Done is set, the programming is complete, then go to step 7.
6. Write the next “free space” number of words of the PSL image to the PSL Programming Port and go to step 3.
7. Reset the PR Request bit in the PSL Programming Control field. The PSL should now be operational.
8. Initialize the PSL state.

10.3.2 Flash Procedures

CAIA-compliant adapters might contain a flash for storing non-volatile information.

Read:

1. Set the flash address (VSEC offset x‘50’) to the first word in the flash to read. The value is a word (4-byte) address, not a byte address.
2. Set the flash size (VSEC offset x‘54’) to the number of 512 word (2 KB) blocks to read.
3. Set the Read Request bit in the Flash Status/Control Register (VSEC offset x‘28’; bit 27) to ‘1’ to start the read operation. The hardware will start reading the flash device and buffering the data.
4. Perform the requested number of reads from the flash data port. The number of reads required is $512 \times$ the value written to flash size.

Program:

1. Set the flash address (VSEC offset x‘50’) to the first block in flash to program. The address must be block aligned or 32K words (lower 15 bits must be ‘0’).
2. Set the flash size (VSEC offset x‘54’) to the number of 32K word blocks (128 KB) blocks to program.
3. Set the Program Request bit in the Flash Status/Control Register (VSEC offset x‘28’; bit 26) to ‘1’ to start the program operation. The hardware will start erasing the requested number of blocks to prepare for programming.
4. Poll the Flash Status/Control Register until the Erase Status is ‘1’ indicating the erase phase is complete and ready for programming.
5. Perform the requested number of writes to the flash data port. The number of writes required is $32K \times$ the value written to flash size.

10.4 Dual Bus Configuration Space

The CAIA allows for a single CAIA-compliant device to support multiple independent PCIe ports: one for commands and data and one for data only. For this type of configuration, the dual-bus PCIe configuration space is presented to the system for the secondary busses. The dual-bus configuration space is not valid for bi-modal devices operating in PCIe mode.

Table 10-5 defines the PCIe type 0 configuration header for the data only port of a dual-bus CAIA-compliant device.

Note: The configuration space is defined in a little endian format to conform to the PCIe standard. This diverges from the other facilities in the CAIA which are defined in a big-endian format.

Table 10-5. PCIe Type 0 Configuration Header (Data Only Port of Dual-Bus CAIA-Compliant Device) (Page 1 of 2)

Field Name	Configuration Header Offset	Bits	Description
Device ID	x‘0’	31:16	The Device ID is an implementation-specific field. The Device ID field is the Device_ID[31:24] (byte 3) concatenated with the Device_ID[23:16] (byte 2).
Vendor ID	x‘0’	15:0	The IBM Vendor ID is an implementation-specific field. The Vendor ID field is the Vendor_ID[31:24] (byte 3) concatenated with the Vendor_ID[23:16] (byte 2).
Status	x‘4’	31:16	See the PCIe 3.0 Specification
Command	x‘4’	15:0	See the PCIe 3.0 Specification
Class Code	x‘8’	31:8	The Class Code is x‘120000’ for all CAPI devices, including bi-modal devices set to CAPI mode. The Class Code is Class_Code[31:24] (byte 3) concatenated with SubClass_Code[23:16] (byte 2) concatenated with Programming_Interface[15:8] (byte 1).
Revision ID	x‘8’	7:0	The Revision ID is an implementation-specific field.
BIST	x‘C’	31:24	The BIST is an implementation-specific field.
Header Type	x‘C’	23:16	Read-only field. Set to x‘00’.
Master latency Timer	x‘C’	15:8	Read-only field. Set to x‘00’.
Cache Line Size	x‘C’	7:0	Read-only field. Set to x‘00’.
Base Address Register 0	x‘10’	31:0	No base address. Read-only field. Always returns a value of x‘00000000’. **Note:** This register is represented as little endian, where the most significant bit is bit 63 and the least significant bit is bit 0.
Base Address Register 1	x‘14’	31:0	No base address. Read-only field. Always returns a value of x‘00000000’. **Note:** This register is represented as little endian, where the most significant bit is bit 63 and the least significant bit is bit 0.
Base Address Register 2	x‘18’	31:0	No base address. Read-only field. Always returns a value of x‘00000000’. **Note:** This register is represented as little endian, where the most significant bit is bit 63 and the least significant bit is bit 0.
Base Address Register 3	x‘1C’	31:0	No base address. Read-only field. Always returns a value of x‘00000000’. **Note:** This register is represented as lttle endian, where the most significant bit is bit 63 and the least significant bit is bit 0.

Table 10-5. PCIe Type 0 Configuration Header (Data Only Port of Dual-Bus CAIA-Compliant Device) (Page 2 of 2)

Field Name	Configuration Header Offset	Bits	Description
Base Address Register 4	x‘20’	31:0	No base address. Read-only field. Always returns a value of x‘00000000’. **Note:** This register is represented as little endian, where the most significant bit is bit 63 and the least significant bit is bit 0.
Base Address Register 5	x‘24’	31:0	No base address. Read-only field. Always returns a value of x‘00000000’. **Note:** This register is represented as lttle endian, where the most significant bit is bit 63 and the least significant bit is bit 0.
Cardbus CIS Pointer	x‘28’	31:0	Read-only field. Set to x‘00’.
Subsystem ID	x‘2C’	31:16	The Subsystem ID is an implementation specific field. The Subsystem ID field is the Subsystem_ID[31:24] (byte 3) concatenated with the Subsystem_ID[23:16] (byte 2).
Subsystem Vendor ID	x‘2C’	15:0	The IBM Subsystem Vendor ID is an implementation specific field. The Subsystem Vendor ID field is the Subsystem_Vendor_ID[31:24] (byte 3) concatenated with the Subsystem_Vendor_ID[23:16] (byte 2).
Expansion ROM Base Address	x‘30’	31:0	Read-only field. Set to x‘00’.
Reserved	x‘34’	31:8	Read-only field. Set to x‘00’.
Capabilities Pointer	x‘34’	7:0	Read-only field. Set to x‘00’.
Reserved	x‘38’	31:0	Read-only field. Set to x‘00’.
Max_Lat	x‘3C’	31:24	Read-only field. Set to x‘00’.
Min_Gnt	x‘3C’	23:16	Read-only field. Set to x‘00’.
Interrupt Pin	x‘3C’	15:8	Read-only field. Set to x‘00’.
Interrupt Line	x‘3C’	7:0	Read-only field. Set to x‘00’.

10.5 PCIe Reset

When a device is operating in CAPI mode, the resets defined by PCIe have the following behaviors:

- Fundamental Reset (PERST)

  Fundamental reset is triggered when PERST is applied. There are two types of fundamental resets: cold and warm. A cold reset is when power is applied to the device, and a warm reset is when PERST is asserted with power already applied to the device.

  Asserting PERST has the same effect for both cold and warm resets. The device is restored to the power-on state. For devices that have a base PCIe mode (that is, a bi-modal devices), asserting PERST resets the device to the power-on reset state. After being reset, the bi-modal device is operating in PCIe mode.

  Following a fundamental reset, the PSL must be downloaded for all CAIA-compliant devices with a loadable PSL.

- Hot Reset (in-band)

  Hot reset is triggered by the TS1 and TS2 training ordered set defined by the PCIe protocol specification.

  When a device receives a hot reset, the PCIe configuration space is reset as defined in the PCIe specification. For bi-modal devices, the CAPI Protocol Enable bit in the Vendor-Specific Extended Capability Structure retains the current value (that is, the CAPI Protocol Enable bit is sticky).

  Following a hot reset, the PSL must be downloaded for all CAIA-compliant devices with a loadable PSL.

10.6 Bi-Modal Device Support

A CAIA-compliant device, that supports only CAPI mode, powers on with the PCIe Type 0 configuration space defined in *Section 10.1* on page 155 and the CAPI Protocol Enable bit set to '1'.

A bi-modal CAIA-compliant device supports the standard PCIe I/O device mode (non-CAPI) and is capable of supporting the coherent CAPI protocol. Bi-modal devices power on with their PCIe mode configuration space and a VSEC structure in the capability list. The CAPI Protocol Enable bit defaults to ‘0’ (CAPI protocol disabled). The class code is set in the PCIe mode configuration space according to the PCIE mode device class (for example, x‘010400’ for a RAID controller).

During initialization, if system software detects the existence of a CAPI VSEC and the adapter is to be used as a CAIA-compliant device, system software sets the VSEC CAPI Enable bit to ‘1’. Before making the determination to enable the device as a PCIe or CAIA-compliant device, system software only performs reads from the PCIe configuration space and VSEC structures.

After the CAPI Protocol Enable bit is set to ‘1’, the bi-modal device presents the PCIe Type 0 configuration space defined in *Section 10.1* on page 155. System software provides the CAIA-compliant device 100 ms after enabling the CAPI protocol before access to the PCIe configuration space. PCIe configuration register settings, that are not specifically defined by the CAIA, are the same as the standard PCIe configuration space settings.

Appendix A. Memory Maps

This appendix contains the mapping of all registers defined by the Coherent Accelerator Interface Architecture (CAIA) in the real address space. As shown in *Table A-1. CAIA-Compliant Processor Memory Map* on page 172, the memory map for a single CAIA-compliant processor is divided into five sections:

- *PSL Privilege 1 Memory Map*
- *PSL Slice Privilege 1 Memory Map*
- *PSL Slice Privilege 2 Memory Map*
- *AFU Descriptor Memory Map*
- *AFU Problem State (per Slice)*

The starting location for the memory map is defined in implementation-dependent base address registers (P1_Base and P2_Base). The base registers are used to calculate the address for all registers. An implementation can place address holes in the memory map (that is, areas where registers are not defined) to simplify decoding of the registers. The base registers for a CAIA-compliant accelerator are implementation dependent. See the specific implementation documentation for more information.

Implementation Note:

A CAIA-compliant accelerator implementation must provide at least two base registers for relocating the internal registers. All base register (P1_Base and P2_Base) values must be initialized during the PCIe initialization sequence. The base registers are set using the standard PCIe BAR registers.

A CAIA-compliant accelerator must respond to the full address range specified by the base registers and the implementation-dependent memory map. An implementation must respond to all address holes. A value of zero must be returned for all read operations, and the data is ignored for all write operations.

Table A-1. CAIA-*Compliant Processor Memory Map*

<table>
<tr><th>Real Address
(Hexadecimal)</th><th>Area</th><th>Description</th></tr>
<tr><td colspan="3">Privilege 1 (P1) Area for PSL
• This area should be mapped with a WIMG setting of ‘0101’.</td></tr>
<tr><td>P1_Base</td><td>Start of P1 area</td><td>Start of the privilege 1 area</td></tr>
<tr><td>P1_Base + x‘FFFF’</td><td>End of P1 area</td><td>End of the privilege 1 area</td></tr>
<tr><td colspan="3">Privilege 1 for each PSL slice (P1(n)) Area; where 0 ≤ n ≤ (maximum number of PSL slices - 1)[1]
• P1(0) ≥ x‘10000’: The starting address of the area.
• P1(n) = P1(0) + (n × x‘100’); where 0 ≤ n ≤ (maximum number of PSL slices - 1)[1].
• This area should be mapped with a WIMG setting of ‘0101’.</td></tr>
<tr><td>P1_Base | P1(n)</td><td>Start of P1(n) area</td><td>Start of the privilege 1 area for PSL_Slice(n)</td></tr>
<tr><td>P1_Base | (P1(n) + x‘FF’)</td><td>End of P1(n) area</td><td>End of the privilege 1 area for PSL_Slice(n)</td></tr>
<tr><td colspan="3">PSL Privilege 2 (P2(n)) Area; where 0 ≤ n ≤ (maximum number of PSL slices - 1)[1]
• P2(n) = (n × x‘1000’); where 0 ≤ n ≤ (maximum number of PSL slices- 1)[1].
• This area should be mapped with a WIMG setting of ‘0101’.</td></tr>
<tr><td>P2_Base | P2(n)</td><td>Start of P2(n) area</td><td>Start of the privilege 2 area for PSL_Slice(n)</td></tr>
<tr><td>P2_Base | (P2(n) + x‘FFFF’)</td><td>End of P2(n) area</td><td>End of the privilege 2 area for PSL_Slice(n)</td></tr>
<tr><td colspan="3">AFU Descriptor (AD(n)) Area; where 0 ≤ n ≤ (maximum number of AFUs - 1)[1]
• AD(0) ≥ P2(max) + X‘10000’; where max = (maximum number of PSL Slices - 1).[4]
• AD(n) = PS(0) + (n × AD_Size[5]); where 0 ≤ n ≤ (maximum number of AFUs - 1)[1].
• This WIMG settings for this area is AFU implementation specific. The typical WIMG setting is ‘0101’.</td></tr>
<tr><td>P2_Base | AD(n)</td><td>Start of PS(n) area</td><td>Start of the problem state area for AFU(n)</td></tr>
<tr><td>P2_Base | (AD(n) + AD_Size[5] - 1)</td><td>End of PS(n) area</td><td>End of the problem state area for AFU(n)</td></tr>
<tr><td colspan="3">AFU Problem State (PS(n)) Area; where 0 ≤ n ≤ (maximum number of AFUs - 1)[1]
• PS(0) ≥ AD(max) + (AD_Size × n); where 0 ≤ n ≤ (maximum number of AFUs - 1)[1], max = (maximum number of PSL Slices - 1).[4]
• PS(n) = PS(0) + (n × PS_Size[3]); where 0 ≤ n ≤ (maximum number of AFUs - 1)[1].
• This WIMG settings for this area are AFU implementation specific. The typical WIMG setting is ‘0101’. Some implementations might support regions of this area to be mapped as not guarded (G = 0) to allow store gathering for better performance.</td></tr>
<tr><td>P2_Base | PS(n)</td><td>Start of PS(n) area</td><td>Start of the problem state area for AFU(n)</td></tr>
<tr><td>P2_Base | (PS(n) + PS_Size[3] - 1)</td><td>End of PS(n) area</td><td>End of the problem state area for AFU(n)</td></tr>
<tr><td colspan="3">Notes:
1. The value n ranges from zero to the number of PSL slices or AFUs minus 1 (0 ≤ n ≤ maximum number of AFUs -1). If the number of AFUs is not a power of 2, an implementation can choose to increase the value of n to the next power of 2 boundary and reserve the extra space. Doing so simplifies decoding the address ranges. There is one PSL_Slice for each AFU.
2. This table assumes that the base starts on a power of 2 boundary that is greater than or equal to the size of the memory map area.
3. The problem state area size (PS_Size) is an implementation-dependent parameter. PSL implementations must consider reserving 32 MB (x‘2000000’) of space for CAPI devices using on-adapter memory. For more information, see the implementation-specific documentation.
4. The problem state area for AFU 0 starts at the next natural boundary for the problem state area reserved for a single AFU slice. For example, the problem state area for a single AFU design with 32 MB of area reserved starts on the next 32 MB boundary from the privileged 2 base.
5. The AFU descriptor size (AD_Size) is an implementation-dependent parameter. The AFU descriptor size is provided in the Vendor Specific Extended Configuration (VSEC). PSL implementations must consider reserving 64 KB (0x100000) of space for each AFU descriptor. For more information, see the implementation-specific documentation and Section 10.3 CAIA Vendor-Specific Extended Capability Structure on page 160.</td></tr>
</table>

A.1 PSL Privilege 1 Memory Map

Table A-2. lists all the CAIA-compliant PSL registers that allow only privilege 1 access.

Table A-2. PSL Privilege 1 Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
Configuration and Control Area			
x‘0000’	PSL_CtxTime	PSL Context Swap Time Register (PSL_CtxTime) (see page 97)[1]	Read/Write
x‘0008’	PSL_ErrIVTE	PSL Error Interrupt Register (PSL_ErrIVTE) (see page 99)	Read/Write
x‘0010’	PSL_KEY1	PSL Key One Register (PSL_KEY1) (see page 100)	Write
x‘0018’	PSL_KEY2	PSL Control Register (PSL_Control) (see page 102)	Write
x‘0020’	PSL_Control	PSL Control Register (PSL_Control) (see page 102)	Write
x‘0028’ - x‘005F’	Reserved	Reserved	
AFU Download Control			
x‘0060’	AFU_DLCNTL	AFU Download Control Register (AFU_DLCNTL) (see page 104)[1]	Read/Write
x‘0068’	AFU_DLADDR	AFU Download Address Register (AFU_DLADDR) (see page 106)[1]	Read/Write
x‘0070’ - x‘007F’	Reserved	Reserved	
PSL Lookaside Buffer Management Area			
x‘0080’	PSL_LBISEL	PSL Lookaside Buffer Invalidate Selection Register (PSL_LBISEL) (see page 107) PID and LPID of the translations to invalidate.	Read/Write
x‘0088’	PSL_SLBIE	PSL SLB Invalidate Entry Register (PSL_SLBIE) (see page 108) Effective segment ID (ESID) of the SLB entry to invalidate.	Read/Write
x‘0090’	PSL_SLBIA	PSL SLB Invalidate All Register (PSL_SLBIA) (see page 110) Invalidate all SLB entries.	Read/Write
x‘0098’	Reserved	Reserved	
x‘00A0’	PSL_TLBIE	PSL TLB Invalidate Entry (PSL_TLBIE) (see page 112) Virtual address (VA) of the TLB entry to invalidate.	Read/Write
x‘00A8’	PSL_TLBIA	PSL TLB Invalidate All (PSL_TLBIA) (see page 114) Invalidate all TLB entries.	Read/Write
x‘00B0’	PSL_AFUSEL	PSL AFU Selection Register (PSL_AFUSEL) (see page 116) AFU slice selection for the translations to invalidate.	Read/Write
x‘00B8’ - x‘00BF’	Reserved	Reserved	
Implementation-Dependent Area. (See the specific implementation documentation for a detailed description of these registers).			
x‘00C0’ - x‘7EFF’	PV1_ImplRegs	Privilege 1 implementation-dependent registers	

1. Implementation of this facility is optional.

Table A-2. PSL Privilege 1 Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
PCIe MSI-X Pending Bit Array (PBS) Area. (Not used. Reserved for future compatibility with PCIe MSI-X Architecture).			
x'7F00' - x'7FFF'	MSI-X_PBA	Reserved area for MSI-X pending bit array (PBA). The PCIe MSI-X extended capability structure points to the start of this location for the pending bit array. This area is not used by the PSL but must be reserved for compatibility with the PCIe standard. One bit per MSI-X table entry (maximum 2048 bits).	
PCIe MSI-X Table Area. (Not used. Reserved for future compatibility with PCIe MSI-X Architecture).			
x'8000' - x'FFFF'	MSI-X_Table	Reserved area for the MSI-X Table. The PCIe MSI-X extended capability structure points to the start of this location for the MSI-X table. This area is not used by the PSL but must be reserved for compatibility with the PCIe standard. MSI-X table entry consist of (maximum 2048 entries): Word 0 = Lower 32-bits of interrupt message address Word 1 = Upper 32-bits of interrupt message address Word 2 = Interrupt data Word 3 = Interrupt vector control	

1. Implementation of this facility is optional.

A.2 PSL Slice Privilege 1 Memory Map

Table A-3. lists all the CAIA-compliant PSL slice registers that allow only privilege 1 access.

Table A-3. PSL Slice Privilege 1 Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
Configuration Area			
x'00'	PSL_SR_An	PSL State Register (PSL_SR_An) (see page 74) for AFU n	Read/Write
x'08'	PSL_LPID_An	PSL Logical Partition ID Register (PSL_LPID_An) (see page 77) for AFU n	Read/Write
x'10'	PSL_AMBAR_An	PSL AFU Memory Base Address Register (PSL_AMBAR_An) (see page 78) for AFU n[1]	Read/Write
x'18'	PSL_SPOffset_An	PSL AFU Scratch Pad Offset Register (PSL_SPOffset_An) (see page 80) for AFU n[1]	Read/Write
x'20'	PSL_ID_An	PSL ID Register (PSL_ID_An) (see page 82) for AFU n	Read/Write
x'28'	PSL_SERR_An	PSL Slice Error Register (PSL_SERR_An) (see page 84) for AFU n	Read/Write
Memory Management Registers			
x'30'	PSL_SDR_An	PSL Storage Description Register (PSL_SDR_An) (see page 86) for AFU n Also see the *PowerPC Architecture, Book III* for a description of this register.	Read/Write
x'38'	PSL_AMOR_An	PSL Authority Mask Override Register (PSL_AMOR_An) (see page 87) for AFU n Also see the *PowerPC Architecture, Book III* for a description of this register.	Read/Write
x'40' - x'5F'	Reserved	Reserved	
Pointer Area			
x'80'	HAURP_An	Hypervisor Accelerator Utilization Record Pointer Register (HAURP_An) (see page 88) for AFU n[1]	Read/Write
x'88'	PSL_SPAP_An	PSL Scheduled Processes Area Pointer Register (PSL_SPAP_An) (see page 89) for AFU n[1]	Read/Write
x'90'	PSL_LLCMD_An	PSL Linked List Command Register (PSL_LLCMD_An) (see page 90) for AFU n[1]	Read/Write
x'98' - x'9F'	Reserved	Reserved	
Control Area			
x'A0'	PSL_SCNTL_An	PSL Slice Control Register (PSL_SCNTL_An) (see page 91) for AFU n	Read/Write
x'A8'	PSL_CtxTime_An	PSL Context Swap Time Slice Register (PSL_CtxTime_An) (see page 93)[1]	Read/Write
x'B0'	PSL_IVTE_Offset_An	PSL IVTE Offset Register (PSL_IVTE_Offset_An) (see page 94) for AFU n	Read/Write
x'B8'	PSL_IVTE_Limit_An	PSL IVTE Limit Register (PSL_IVTE_Limit_An) (see page 95) for AFU n	Read/Write
Implementation-Dependent Area. (See the specific implementation documentation for a detailed description of these registers).			
x'C0' - x'FF'	PV1n_ImplRegs	Privilege 1 implementation-dependent registers	

1. Implementation of this facility is optional.

A.3 PSL Slice Privilege 2 Memory Map

Table A-4. lists all the CAIA-compliant PSL slice registers that allow only privilege 2 access.

Table A-4. PSL Slice Privilege 2 Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
Configuration and Control Area			
x'0000'	PSL_PID_TID_An	PSL Process and Thread Identification Register (PSL_PID_TID_An) (see page 118) for AFU n	Read/Write
x'0008'	CSRP_An	Context Save/Restore Pointer Register (CSRP_An) (see page 119) for AFU n[1]	Read/Write
x'0010'	AURP0_An	Accelerator Utilization Record Pointer Zero Register (AURP0_An) (see page 120) for AFU n[1]	Read/Write
x'0018'	AURP1_An	Accelerator Utilization Record Pointer One Register (AURP1_An) (see page 121) for AFU n[1]	Read/Write
x'0020'	SSTP0_An	Storage Segment Table Pointer Zero Register (SSTP0_An) (see page 123) for AFU n	Read/Write
x'0028'	SSTP1_An	Storage Segment Table Pointer One Register (SSTP1_An) (see page 125) for AFU n	Read/Write
x'0030'	PSL_AMR_An	PSL Authority Mask Register (PSL_AMR_An) (see page 126) for AFU n Also see the *PowerPC Architecture, Book III* for a description of this register.	Read/Write
x'0038'	Reserved	Reserved	Reserved
Segment Lookaside Buffer Management Registers			
x'0040'	SLBIE_An	SLB Invalidate Entry Register (SLBIE_An) (see page 128) For AFU n Effective segment ID (ESID) of the SLB entry to invalidate.	Read/Write
x'0048'	SLBIA_An	SLB Invalidate All Register (SLBIA_An) (see page 130) For AFU n Invalidate all SLB entries.	Read/Write
x'0050'	SLBI_Select_An	SLB Invalidate All Register (SLBIA_An) (see page 130) For AFU n Lookaside buffer invalidate selector.	Read/Write
x'0058' - x'005F'	Reserved	Reserved	Reserved
Interrupt Registers			
x'0060'	PSL_DSISR_An	PSL Data Storage Interrupt Status Register (PSL_DSISR_An) (see page 133) for AFU n Also see the *PowerPC Architecture, Book III* for a description of this register.	Read Only
x'0068'	PSL_DAR_An	PSL Data Address Register (PSL_DAR_An) (see page 134) for AFU n Also see the *PowerPC Architecture, Book III* for a description of this register.	Read Only
x'0070'	PSL_DSR_An	PSL Data Segment Register (PSL_DSR_An) (see page 135) for AFU n Also see the *PowerPC Architecture, Book III* for a description of the fields in this register.	Read Only
x'0078'	PSL_TFC_An	PSL Translation Fault Control Register (PSL_TFC_An) (see page 136) for AFU n	Read/Write
x'0080'	PSL_PEHandle_An	PSL Process Element Handle Register (PSL_PEHandle_An) (see page 138) for AFU n[1]	Read Only

1. Implementation of this facility is optional.

Table A-4. PSL Slice Privilege 2 Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
x‘0088’	PSL_ErrStat_An	PSL Error Status Register (PSL_ErrStat_An) (see page 139) for AFU n	Read/Write
AFU Registers			
x‘0090’	AFU_Cntl_An	AFU Control Register (AFU_Cntl_An) (see page 140) for AFU n	Read/Write
x‘0098’	AFU_ERR_An	AFU Error Register (AFU_ERR_An) (see page 142) for AFU n	Read Only
Work Element Descriptor			
x‘00A0’	PSL_WED_An	PSL WED Register (PSL_WED_An) (see page 143) for AFU n	Read/Write
x‘00A8’ - x‘00BF’	Reserved	Reserved	Reserved
Reserved			
x‘00C0’ - x‘00FF’	Reserved	Reserved	Reserved
Implementation-Dependent Area. (See the specific implementation documentation for a detailed description of these registers.)			
x‘0100’ - x‘FFFF’	PV2_ImplRegs	Privilege 2 implementation-dependent registers	

1. Implementation of this facility is optional.

A.4 AFU Descriptor Memory Map

Table A-5. lists all the required AFU descriptor registers for a CAIA-compliant device.

Table A-5. AFU Descriptor Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
AFU Descriptor Header			
x'0'	num_ints_per_process	AFU Descriptor Format (see page 147) for AFU n[1]	Read Only
x'2'	num_of_processes		Read/Write
x'4'	num_of_afu_CRs		Read Only
x'6'	req_prog_model		Read Only
x'8' - x'1F'	Reserved	Reserved	Reserved
AFU Configuration Record Header			
x'20'	AFU_CR_len	AFU Descriptor Format (see page 147) for AFU n[1]	Read Only
			Read Only
x'22' - x'27'	Reserved	Reserved	Reserved
x'28'	AFU_CR_offset	AFU Descriptor Format (see page 147) for AFU n[1]	Read Only
AFU Problem State Area Control			
x'30'	PerProcessPSA_control	AFU Descriptor Format (see page 147) for AFU n[1]	Read Only
x'31'	PerProcessPSA_length		Read Only
x'38'	PerProcessPSA_offset	AFU Descriptor Format (see page 147) for AFU n[1]	Read Only
AFU Error Buffer			
x'40'	AFU_EB_len	AFU Descriptor Format (see page 147) for AFU n[1]	Read Only
x'48'	AFU_EB_offset		Read Only
AFU Configuration Record Area			
AFU_CR_offset	AFU_CR_Area_Start	Start of AFU Configuration Record area. See AFU Descriptor Format (see page 147) for AFU n[1]	Read/Write
AFU_CR_offset + (num_of_afu_CRs * AFU_CR_len) - 1	AFU_CR_Area_End		

1. All accesses to these registers must be either 32-bit or 64-bit operations.

Appendix B. Examples

This appendix contains examples of how a CAIA-compliant device can be used by applications and system software. The examples provided in this section are:

- *Dedicated Process Example*
- *Virtualization Example*
- *Interrupt Examples*
- *Translation Fault Handling*
- *AFU Deallocate Sequence Example*
- *AFU Recovery Sequence Example*

B.1 Dedicated Process Example

The simplest programming model for a CAIA-compliant device is the dedicated process programming model. Under this model, the system administrator assigns the accelerator function unit slice (AFU slice) for exclusive use by a single application. In this model, the scheduled processes area and context manager in the PSL are not used.

This section provides an example of how a system might implement the dedicated-process programming model.

B.1.1 Allocation and Initialization Procedure

A high-level procedure on how an AFU might be assigned to an application in a managed system follows:

1. System administrator assigns a slice of a CAIA-compliant device to a logical partition (that is a single operating-system image). In the case of a single-slice implementation or where all slices are assigned to the same operating-system image, the full CAIA-compliant device will be assigned to the logical partition.
2. Hypervisor initializes the state of the PSL for the operating-system context.
   - PCIe Configuration Space to assign MMIO region to operating system
   - PSL Slice Control Register (PSL_SCNTL_An) to set the programming model
   - Interrupt Vector Table Entry Offset (PSL_IVTE_Offset_An)
   - Interrupt Vector Table Entry Limit (PSL_IVTE_Limit_An)
   - PSL State Register (PSL_SR_An)
   - PSL Logical Partition ID Register (PSL_LPID_An)
   - PSL Storage Description Register (PSL_SDR_An)
   - PSL Authority Mask Override Register (PSL_AMOR_An)
3. Hypervisor builds a device tree for the AFU slice (or CAIA-compliant device).
   - Contains information about the CAIA-compliant device (for example, MMIO base, logical interrupt numbers, and so on).
4. Application or user library performs system calls to:
   - Query available acceleration resources.
   - Create and register event handlers for AFU interrupts.
   - Get user-space MMIO base.
   - Download and start an accelerator with a pointer to the application's job queue (that is, the WED).
     Note: The job queue is application dependent and not defined by the CAIA.

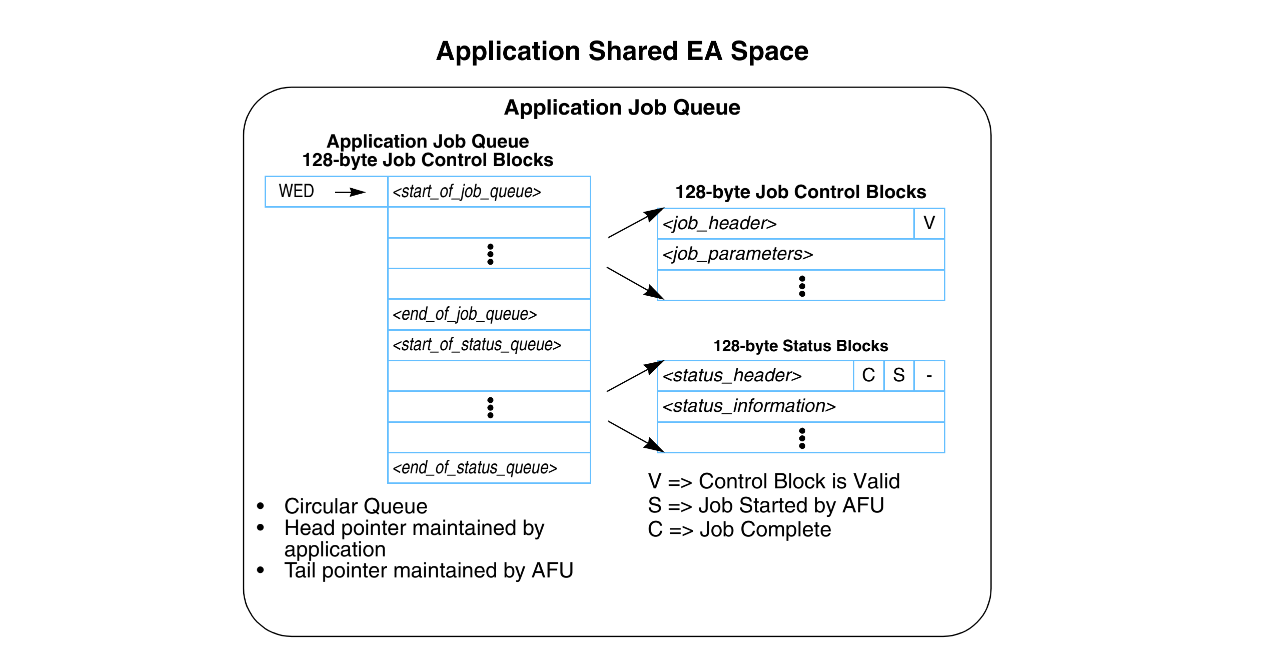
5. Operating system initializes the state of the PSL for the application's process and starts the AFU.
   - PSL Process and Thread Identification Register (PSL_PID_TID_An)
   - Virtual address (VA) Storage Segment Table Pointer Zero Register (SSTP0_An) and Storage Segment Table Pointer One Register (SSTP1_An)
   - PSL Authority Mask Register (PSL_AMR_An)
   - PSL Work Element Descriptor PSL WED Register (PSL_WED_An)
6. AFU is now running and monitoring the application's job queue.

B.1.2 User Job Initiation

The CAIA does not define how job initiation should be performed. Job initiation is application specific. The following example is a very high-level example on how an application might initiate a job on the accelerator and receiving completion status.

In this example, the work element descriptor provided to the operating system during the allocation and initialization sequence is an effective address (EA) pointer to a job queue located in the application's memory space. *Figure B-1.* illustrates the job and completion queue.

Figure B-1. Application Job Queue (User)



After initialization, the head and tail pointers for the job and status queues are set to the start of the corresponding queue, the length of the circular queues are set, and the headers for each queue entry are set to zero. The occupancy of both queues is controlled by the job header. The job queue and status queue contain the same number of entries and are allocated in pairs.

To initiate a new job, the application first checks the status headers at the tail of the queue to ensure a job control/status block pair is available (that is the valid bit (V) in the job header is zero). It then fills in the job parameters and sets the valid bit (V) in the job control block. The application's tail pointer is then moved to the next control/status block pair, wrapping back to the start of the job queue if necessary.

Note: To prevent the AFU from wrapping around the queue and starting a job that has already been completed, the application ensures that the tail pointer points to an invalid job control block after initiating a job. This implies there must be two or more control/status block pairs available in the queues before a new job can be initiated.

After being started, the AFU begins monitoring the job header at the head pointer for work to be performed. When the valid bit (V) in the job header is set, the AFU reads the full control block, starts the acceleration job, and sets the started bit (S) in the corresponding status header. When the acceleration job is complete, the AFU sets the complete bit (C) in the status header, moves the head pointer to the next control bock (wrapping back to the start of the job queue if necessary), and begins monitoring the job header for the “V” bit to be set.

Jobs are not required to complete in queue order. Applications monitor the completion bits in the status headers for job completion. The application can request a completion notification from the AFU (either an interrupt or a write to a shared cache line to wake a processor thread) to avoid a software polling loop. When a job is complete, the application reads the status information, clears the “V” bit in the corresponding job header, and sets the status header to zero (in that order).

B.2 Virtualization Example

The PSL's context manager provides a mechanism to enable sharing of an accelerator function unit (AFU) by more than one application. In the shared virtualization programming models, a linked list of scheduled processes provides the process state for all applications registered to use the AFU. When operating in the shared virtualization model, the AFU is time-sliced between all registered processes.

B.2.1 Shared Model Allocation and Initialization Procedure

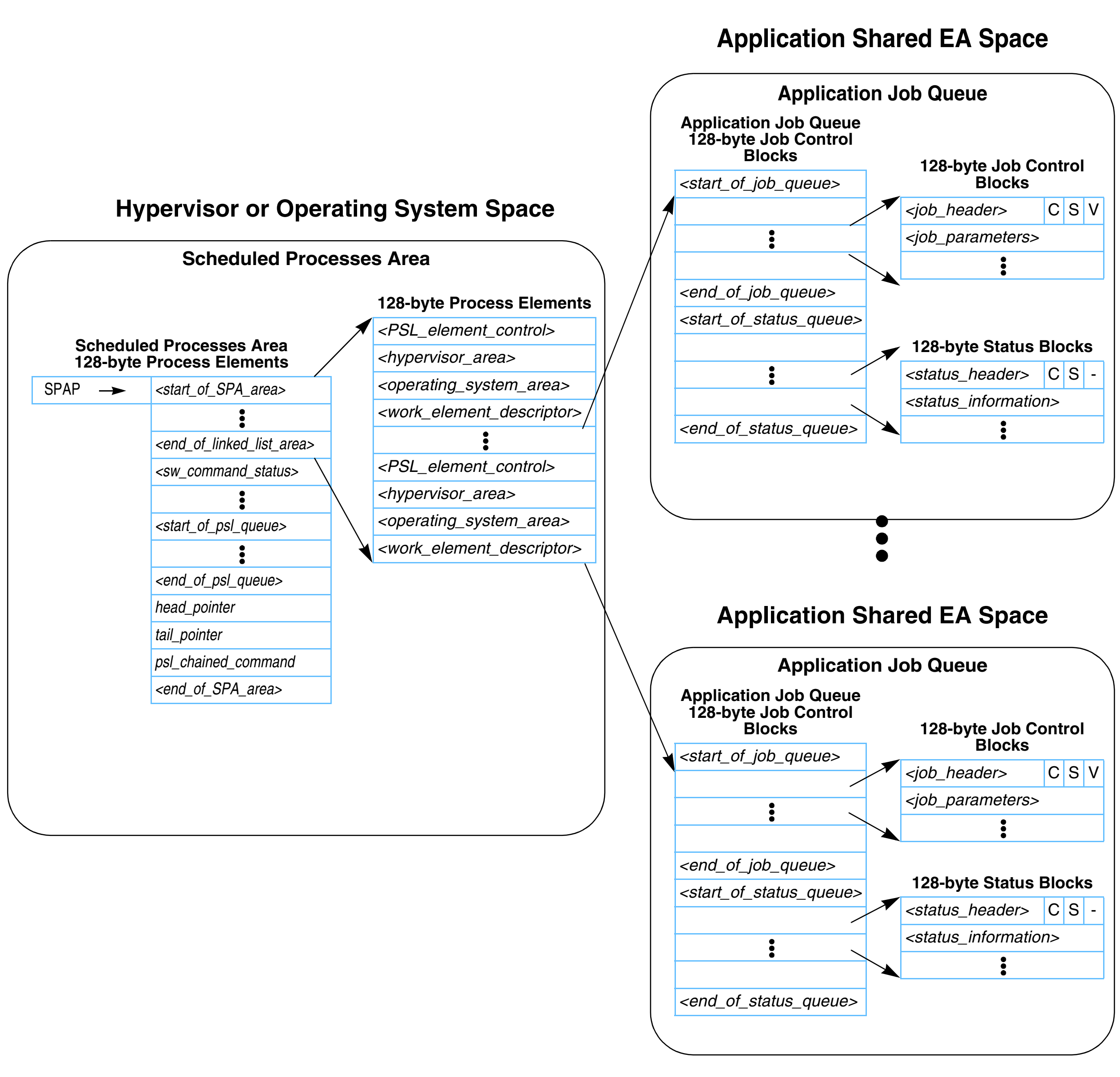
A high-level procedure for how an AFU might be assigned to a hypervisor for sharing by more than one partition in the system follows:

1. System administrator assigns a slice of a CAIA-compliant device to the hypervisor and defines the function. In the case of a single-slice implementation or where all slices are assigned to the hypervisor, the full CAIA-compliant device is assigned.
2. Hypervisor builds a device tree for the AFU slice (or CAIA-compliant device) and sets the programming model.
   - Contains information about the CAIA-compliant device (for example, acceleration function).
   - PSL Slice Control Register (PSL_SCNTL_An) to set the programming model
3. Application or user library performs system calls to:
   - Query available acceleration functions.
   - Create and register event handlers for AFU interrupts
   - Register with the accelerator and provide a pointer to the application's job queue (that is, the WED).
     Note: The job queue is application dependent and not defined by the CAIA.
4. Operating system performs an HCALL to register the application's process state with associated accelerator function.
   - PSL Process and Thread Identification
   - Virtual address (VA) Storage Segment Table Pointer
   - PSL Interrupt Vector Table Entries
   - PSL Authority Mask
   - PSL Work Element Descriptor
5. Hypervisor creates an entry in the linked list of scheduled processes containing the process state for the application.
   - See process element for required parameters in *Section 2.3.1 Process Element Entry* on page 33.
6. After a process element is created, the PSL begins giving a time slice of the AFU to the application. When the application's context is running, the AFU monitors the application's job queue for work to be performed.

B.2.2 Context and Job Initiation

When process elements are added to the linked list of scheduled processes, the PSL reads the next element from the linked list, initializes the required state based on the programming model, and starts the AFU with the associated work element descriptor. The PSL allows the AFU to run with the current application's state until either the AFU yields the remainder of its context interval or the context interval expires. At that point, the PSL requests the AFU to save context if required. If the process is not complete, the PSL places the process element into the PSL queue and starts the next context. *Figure B-2.* illustrates the job and completion queue.

Figure B-2. Application Job and Completion Queue



The CAIA does not define how an application initiates jobs in a virtualized model. Job initiation is application specific. See *Appendix B.1.2 User Job Initiation* on page 180 for an example of how an application might initiate a job.

B.3 Interrupt Examples

In the PCIe architecture, MSI-X interrupts are presented to the host processor using a PCIe posted write operation. The address and data for the posted write operation are programmable using a MSI-X table in the MMIO space. The CAIA supports three modes for selecting the address and data for the MSI-X interrupt: fixed platform-specific address, single MSI-X table entry, and full MSI-X table.

Interrupts in the CAIA are presented to the host processor using a mechanism layered on top of the PCIe MSI-X interrupts. The main difference between the PCIe architecture and CAIA for interrupts is the source of the address and data for the posted write operation.

For a CAIA-compliant device operating with a fixed platform-specific address or a single MSI-X entry, the address is calculated using the PSL_IVTE_Offset_An and PSL_IVTE_Limit_An Registers, and the address provided in entry zero of the MSI-X table. The data is always zero for this mode. The PSL_IVTE_Offset_An and PSL_IVTE_An Registers are either set directly using an MMIO in a dedicated-process model or sourced from the process element in a virtualization model. The following equations are used to generate the address for the PCIe posted write operation from the AFU LISNs.

Note: For PSL interrupts reported in the PSL_DSISR_An Register, the LISN is always zero. For PSL error interrupts, the IVTE is sourced directly from the PSL_ErrIVTE Register. The AFU_LISN must be between the range 1 ≤ AFU_LISN ≤ (Max_Ints - 1). The maximum number of interrupts (Max_Ints) is the sum of the interrupt ranges (PSL_IVTE_Limit_An[IVTE_Range_x]).

```
IVTE = PSL_IVTE_Offset_An[IVTE_Offset_0] + AFU_LISN;
         when (1 ≤ AFU_LISN < PSL_IVTE_Limit_An[IVTE_Range_0])
         AND   (PSL_IVTE_Limit_An[IVTE_Range_0] != 0)
       else

IVTE = PSL_IVTE_Offset_An[IVTE_Offset_1] + (AFU_LISN - PSL_IVTE_Limit_An[IVTE_Range_0]);
         when ( (PSL_IVTE_Limit_An[IVTE_Range_0])
                ≤ AFU_LISN <
                  (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1]) )
         AND   (PSL_IVTE_Limit_An[IVTE_Range_1] != 0)
       else

IVTE = PSL_IVTE_Offset_An[IVTE_Offset_2] + ( AFU_LISN -
       (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1]) );
         when ( (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1])
                ≤ AFU_LISN <
                (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1] +
                 PSL_IVTE_Limit_An[IVTE_Range_2]) )
         AND   (PSL_IVTE_Limit_An[IVTE_Range_2] != 0)
       else

IVTE = PSL_IVTE_Offset_An[IVTE_Offset_3] + ( AFU_LISN -
       (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1] +
       PSL_IVTE_Limit_An[IVTE_Range_2]) );
         when ( (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1] +
```

```
                PSL_IVTE_Limit_An[IVTE_Range_2])
              ≤ AFU_LISN <
               (PSL_IVTE_Limit_An[IVTE_Range_0] + PSL_IVTE_Limit_An[IVTE_Range_1] +
                PSL_IVTE_Limit_An[IVTE_Range_2] + PSL_IVTE_Limit_An[IVTE_Range_3]) )
        AND   (PSL_IVTE_Limit_An[IVTE_Range_3] != 0)
      else

No IVTE is generated and No Interrupt is sent for the AFU.

Single MSI-X Table Entry:
   PCIe_MSI-X_address = (MSI-X_Base | (x‘00000000000’ || IVTE || x‘0’)
   PCIe_MSI-X_data = 0
        where the MSI-X_Base is either x‘1000000000000000’ for platform specific
        fixed address or the address from MSI-X_Table_entry[0].

Full MSI-X Table:
   PCIe_MSI-X_address = MSI-X_Table_entry[IVTE]
   PCIe_MSI-X_data = MSI-X_Table_entry[IVTE]
```

Note: For POWER Systems, the MSI-X_Base is x‘1000000000000000’. System software should set the address in entry zero of the MSI-X table to the base address for MSI-X interrupts.

Note: For PSL interrupts reported in the PSL_DSISR_An Register, the IVTE is PSL_IVTE_Offset_An[IVTE_Offset_0] (an LISN of zero). For PSL error interrupts, the IVTE is sourced directly from the PSL_ErrIVTE Register. The AFU_LISN must be between the range 1 ≤ AFU_LISN < (Max_Ints - 1). The maximum number of interrupts (Max_Ints) is the sum of the interrupt ranges (PSL_IVTE_Limit_An[IVTE_Range_x]).

The MSI-X address is used by the POWER processor to create an interrupt source number (ISN) and an offset into an interrupt vector table (IVT_Offset). The IVT_Offset selects an interrupt vector table entry (IVTE) located in system memory that contains the interrupt priority and the destination server number. The destination server number defines the destination server or group of servers that should handle the interrupt. The equations for generating the ISN and the IVT_Offset are as follows:

```
IVT_Offset = (PCIe_MSI-X_address[44..63] | (IVTE || x‘0’) ); for big endian numbering
             (PCIe_MSI-X_address[19..0] | (IVTE || x‘0’) ); for little endian numbering

ISN =        PCIe_MSI-X_address[44..59] (big endian numbering)
             PCIe_MSI-X_address[19..4] (little endian numbering)
```

In a POWER system, the interrupt vector table (IVT) in memory can be a maximum of 2^{20} bytes in length with each entry being 16 bytes (or 2^{16} total entries). The PSL_IVTE_Offset_An and PSL_IVTE_Limit_An Registers provide the hypervisor with a mechanism for limiting an operating system to a range of entries in the IVT.

B.4 Translation Fault Handling

A very high-level description of the translation faults that exist and how they might be handled by hardware and software follows. Listed are the types of translation faults.

- Page Fault – Requested data is not in memory.
- Mapping Fault – Requested data is in memory but not mapped in the page table.
- Protection Fault – Access to the requested data is not allowed by the protection mechanisms.
- Segment Table Pointer Fault – Translation for the virtual segment table pointer is not in the hardware page table.
- Accelerator Utilization Record Pointer Fault – Translation for the virtual accelerator utilization record pointer is not in the hardware page table.
- Data Segment Fault – Segment for requested data not in the segment table.

B.4.1 Page Fault

A page fault occurs when the requested data is not located in memory. Below are the steps leading up to a page fault and how the fault is resolved. A page fault is in the data storage class of faults.

1. The PSL successfully converts the effective address (EA) to a virtual address (VA).
2. The PSL is unable to locate a hardware page table entry for converting the VA into a real address (RA).
3. The PSL sets the appropriate state in the PSL_DSISR_An Register (PSL_DSISR_An[DM,M]) and sends an interrupt to the associated operating system using LISN0.
4. The operating system determines that the page is not resident in memory and writes the PSL_TFC_An[C] bit to a '1' to indicate the translation is not resolved.
5. The PSL sends an acknowledgment to the AFU indicating that the requested virtual memory is paged out and not available at this time.
6. The AFU sends a notification either through pinned shared memory or an interrupt to the corresponding application requesting that the page be touched to cause it to be paged back into memory.
7. The AFU either saves the context and yields the remaining context interval or continues with another request.
8. The application performs a touch of the page that causes a page fault and invokes the normal operating system paging mechanism.
9. When the page fault is resolved (that is, the requested page is in memory), the application notifies the AFU to continue with the corresponding request.

   Note: A possible implementation is for the operating system to schedule the page to be re-installed when receiving the PSL's interrupt. In this case, there needs to be a mechanism for the operating system to notify the corresponding process so that the AFU can be notified to retry the operation. The definition of this mechanism is outside the scope of the CAIA.

B.4.2 Mapping Fault

A mapping fault occurs when the translation for the requested data is not in the hardware page table, but the data is resident in memory. The steps leading up to a mapping fault and how the fault is resolved follows. A mapping fault is in the data storage class of faults.

1. The PSL successfully converts the effective address (EA) to a virtual address (VA).
2. The PSL is unable to locate a hardware page table entry for converting the VA into a RA.
3. The PSL sets the appropriate state in the PSL_DSISR_An Register (PSL_DSISR_An[DM,M]) and sends an interrupt to the associated operating system using LISN0.
4. The operating system determines that the page is resident in memory, but a mapping is not in the hardware page table.
5. The operating system updates the hardware page table with a mapping to the requested page and writes the PSL_TFC_An[R] bit to a ‘1’ indicating that the PSL should restart the corresponding operation.
6. The PSL restarts the operation and translates the EA to an RA. The PSL should find the mapping for the hardware page table.

B.4.3 Protection Fault

A protection fault occurs when the operation being performed to the requested memory is not permitted by the page protection mechanisms; page protection and key protection. A protection fault is in the data storage class of faults. A protection fault can either be an indication to the operating system to perform a “copy on write” or an indication that the application is performing an operation that is not allowed.

A “copy on write” occurs when the content of the page needs to be copied when a write is performed by a thread. This typically occurs after a new process has been forked. In this case, the operating system initially marks the page as read-only and only creates a new page when a write is performed. The following steps lead up to a “copy on write” protection fault and describe how the fault is resolved.

1. The PSL successfully converts the effective address (EA) to a virtual address (VA).
2. The PSL locates a hardware page table entry for the requested page.
3. The PSL checks the page protection and key protection in the page table entry and determines a write operation is being performed to a page marked as read-only and the operation is allowed by the key protection.
4. The PSL sets the appropriate state in the PSL_DSISR_An Register (PSL_DSISR_An[DM,P,S]) and sends an interrupt to the associated operating system using LISN0.
5. The operating system determines the page is resident in memory and that a copy must be performed before allowing the write operation.
6. The operating system copies the page data to another real memory location, updates the hardware page table with a mapping to the new page, and writes the PSL_TFC_An[R] bit to a ‘1’ to indicate the PSL should restart the corresponding operation.
7. The PSL restarts the operation and translates the EA to a RA. The PSL should find the new mapping to allow the write operation.

When the access being performed by the AFU is not allowed by either the page protection or key protection in the hardware page table, the steps are as follows:

1. The PSL successfully converts the effective address (EA) to a virtual address (VA).
2. The PSL locates a hardware page table entry for the requested page.
3. The PSL checks the page protection and key protection in the page table entry and determines the operation being performed is not allowed.
4. The PSL sets the appropriate state in the PSL_DSISR_An Register (PSL_DSISR_An[DM,P,K]) and sends an interrupt to the associated operating system using LISN0.
5. The operating system determines that the AFU is not allowed to perform the requested operation.
6. The operating system purges the current running process from the PSL and sends a signal to the corresponding application running in the host.
7. The operating system restarts the PSL by writing the PSL_TFC_An[A] bit to a ‘1’.
8. The PSL will resume with a new process from the schedule processes area if operating in a virtualized model or remain idle if operating in a dedicated-process model.

B.4.4 Segment Table Pointer Fault

A segment table pointer fault occurs when the translation for the virtual address pointer to the segment table is not in the hardware page table. This fault is basically treated just like any other data storage fault outlined in *Section B.4.1* on page 186, *Section B.4.2* on page 187, and *Section B.4.3* on page 187. The difference between the segment table pointer fault and the other data storage faults is that the PSL does not perform an EA-to-VA translation. This difference eliminates step one in the procedures outlined in the referenced sections.

B.4.5 Accelerator Utilization Record Pointer Fault

An accelerator utilization record pointer fault occurs when the translation for the virtual address pointer to the accelerator utilization record is not in the hardware page table. This fault is basically treated just like any other data storage fault outlined in *Section B.4.1* on page 186, *Section B.4.2* on page 187, and *Section B.4.3* on page 187. The difference between the accelerator utilization record pointer fault and the other data storage faults is the PSL does not perform an EA-to-VA translation. This difference eliminates step one in the procedures outlined in the referenced sections.

B.4.6 Data Segment Fault

A data segment fault occurs when the segment for the requested data is not in the hardware segment table. The steps leading up to a data segment fault and how the fault is resolved are as follows:

1. The PSL is unable to locate a segment in the hardware segment table for converting the EA to a VA.
2. The PSL sets the appropriate state in the PSL_DSISR_An Register (PSL_DSISR_An[DS]) and sends an interrupt to the associated operating system using LISN0.
3. The operating system updates the hardware segment table with the requested segment, issues a **sync** instruction, and then writes the PSL_TFC_An[R] bit to a ‘1’ to indicate that the PSL should restart the corresponding operation.
4. The PSL restarts the operation and translates the EA-to-RA. The PSL should find the mapping for the hardware segment table.

B.5 AFU Deallocate Sequence Example

The following is a high-level description of the procedure that system software should follow when deallocating an AFU. The following procedure assumes that system software is servicing faults from the PSL during the procedure.

1. **Disable AFU**
    a. Write the AFU_CNTL_An[RA] = ‘1’
    b. Wait for AFU_CNTL_An[RS] = ‘10’
2. **Suspend PSL**
    a. Write the PSL_SCNTL_An[Sc] = ‘1’
    b. Wait for PSL_SCNTL_An[Ss] = ‘11’
3. **Purge PSL**
    a. Write the PSL_SCNTL_An[Pc] = ‘1’
    b. Wait for PSL_SCNTL_An[Ps] = ‘11’
4. **Invalidate TLBs**
    Note: This must be performed by the hypervisor.
    a. Write the PSL_LBISEL Register with the PID and LPID (optional depending on the value of PSL_TLBIA[IQ].
    b. EIEIO
    c. Write PSL_TLBIA[IQ] = ‘00’ or ‘10’ or ‘11’ depending on how selective you want to be on TLB invalidates.
    d. Wait until the TLBs have been invalidated.
5. **Invalidate SLBs**
    Note: This can also be done by the hypervisor.

    If Hypervisor

    a. Write the PSL_LBISEL Register with the PID and LPID (optional depending on the value of PSL_SLBIA[IQ]).
    b. EIEIO
    c. Write PSL_SLBIA[IH, IQ] depending on how selective you want to be on SLB invalidates.
    d. Wait until the SLBs have been invalidated.

    _If OS_

    a. Write the SLIBA_An[IH, IQ] depending on how selective you want to be on SLB invalidates.

        Note: The LPID and PID should already be valid.

    b. Wait until the SLBs have been invalidated.
6. **Set the PSL State to require a download of an AFU before allowing an AFU to be enabled.**
    a. Write the PSL_SR_An[CR] both to ‘1’.
7. **Set PSL to Normal operation**
    a. Write the PSL_SCNTL_An[Ps, Pc] both to ‘0’.
    b. Wait until the PSL_SCNTL_An[Ss, Sc] fields are both ‘00’.

B.6 AFU Recovery Sequence Example

A high-level description of the procedure that the hypervisor should follow when recovering an AFU when an operating system has terminated without releasing the AFU is as follows:

1. **Disable the AFU.**
   a. Write the AFU_CNTL_An[RA] = ‘1’
   b. Wait for AFU_CNTL_An[RS] = ‘10’
2. **Purge the PSL.**
   a. Write the PSL_SCNTL_An[Pc] = ‘1’
   b. Wait for PSL_SCNTL_An[Ps] = ‘11’
   While waiting for the purge to complete, the hypervisor must respond to all interrupts reported with the PSL_DSISR_An with either acknowledge or continue, depending on the interrupt type.
3. **Invalidate the TLBs.**
   a. Write the PSL_LBISEL Register with the PID and LPID (optional depending on the value of PSL_TLBIA[IQ].
   b. EIEIO
   c. Write PSL_TLBIA[IQ] = ‘00’ or ‘10’ or ‘11’ depending on how selective you want to be on TLB invalidates.
   d. Wait until the TLBs have been invalidated.
4. **Invalidate the SLBs.**
   a. Write the PSL_LBISEL Register with the PID and LPID (optional depending on the value of PSL_SLBIA[IQ].
   b. EIEIO
   c. Write PSL_SLBIA[IH, IQ] depending on how selective you want to be on SLB invalidates.
   d. Wait until the SLBs have been invalidated.
5. **Set the PSL state to require a download of an AFU before allowing an AFU to be enabled.**
   a. Write the PSL_SR_An[CR] both to ‘1’.
6. **Set the PSL to normal operation.**
   a. Write the PSL_SCNTL_An[Ps, Pc] both to ‘0’.
   b. Wait until the PSL_SCNTL_An[Ss, Sc] fields are both ‘00’.

Glossary

AFE	Accelerator function environment.
AFU	Accelerator function unit.
architecture	A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible implementations.
AUI	Accelerator unit interface.
big endian	A byte-ordering method in memory where the address n of a word corresponds to the most-significant byte. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most-significant byte. See little endian.
cache	High-speed memory close to a processor. A cache usually contains recently-accessed data or instructions, but certain cache-control instructions can lock, evict, or otherwise modify the caching of data or instructions.
caching inhibited	A memory update policy in which the cache is bypassed, and the load or store is performed to or from system memory. A page of storage is considered caching inhibited when the 'I' bit has a value of '1' in the page table. Data located in caching inhibited pages cannot be cached at any memory hierarchy that is not visible to all processors and devices in the system. Stores must update the memory hierarchy to a level that is visible to all processors and devices in the system.
CAIA	Coherent Accelerator Interface Architecture. Defines an architecture for loosely coupled coherent accelerators. The Coherent Accelerator Interface Architecture provides a basis for the development of accelerators coherently connected to a POWER processor.
CAPI	Coherent Accelerator Processor Interface.
coherence	Refers to memory and cache coherence. The correct ordering of stores to a memory address, and the enforcement of any required cache write-backs during accesses to that memory address. Cache coherence is implemented by a hardware snoop (or inquire) method, which compares the memory addresses of a load request with all cached copies of the data at that address. If a cache contains a modified copy of the requested data, the modified data is written back to memory before the pending load request is serviced.
CRC	Cyclic redundancy check.
DAR	Data Address Register.
DMA	Direct memory access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.
EA	Effective address. An address generated or used by a program to reference memory. A memory-management unit translates an effective address to a virtual address, which it then translates to a real address (RA) that accesses real (physical) memory. The maximum size of the effective-address space is 2^{64} bytes.

ECC	Error correction code. A code appended to a data block that can detect and correct bit errors within the block.
ERAT	Effective-to-real-address translation, or a buffer or table that contains such translations, or a table entry that contains such a translation.
ESID	Effective segment ID.
exception	An error, unusual condition, or external signal that can alter a status bit and will cause a corresponding interrupt, if the interrupt is enabled. See interrupt.
fetch	Retrieving instructions from either the cache or system memory and placing them into the instruction queue.
guarded	Prevented from responding to speculative loads and instruction fetches. The operating system typically implements guarding, for example, on all I/O devices.
HAUR	Hypervisor accelerator utilization record.
HCA	Hot/cold assist.
hypervisor	A control (or virtualization) layer between hardware and the operating system. It allocates resources, reserves resources, and protects resources among (for example) sets of AFUs that might be running under different operating systems.
implementation	A particular processor that conforms to the architecture but might differ from other architecture-compliant implementations for example in design, feature set, and implementation of optional features.
INT	Interrupt. A change in machine state in response to an exception.
ISA	Instruction set architecture.
ISN	Interrupt source number.
IVT	Interrupt vector table.
IVTE	Interrupt vector table entry.
KB	Kilobyte.
LA	A local storage (LS) address of a PSL list. It is used as a parameter in a PSL command.
LISN	Logical interrupt source number.
little endian	A byte-ordering method in memory where the address *n* of a word corresponds to the least-significant byte. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the most-significant byte. See big endian.
LISN	Logical interrupt source number.
LPAR	Logical partitioning. A function of an operating system that enables the creation of logical partitions.
LPC	Lowest point of coherency.
LPID	logical-partition identity.

LSb	Least-significant bit. The bit of least value in an address, register, data element, or instruction encoding.
main storage	The effective-address space. It consists physically of real memory (whatever is external to the memory-interface controller), local storage, memory-mapped registers and arrays, memory-mapped I/O devices, and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files.
mask	A pattern of bits used to accept or reject bit patterns in another set of data. Hardware interrupts are enabled and disabled by setting or clearing a string of bits, with each interrupt assigned a bit position in a mask register.
MB	Megabyte.
memory coherency	An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.
memory mapped	Mapped into the Coherent Attached Accelerator's addressable-memory space. Registers, local storage (LS), I/O devices, and other readable or writable storage can be memory-mapped. System software does the mapping.
MMIO	Memory-mapped input/output. See memory mapped.
MMU	Memory management unit. A functional unit that translates between effective addresses (EAs) used by programs and real addresses (RAs) used by physical memory. The MMU also provides protection mechanisms and other functions.
MSb	Most-significant bit. The highest-order bit in an address, registers, data element, or instruction encoding.
MSI	Message signaled interrupt.
MSR	Machine State Register.
no-op	No-operation. A single-cycle operation that does not affect registers or generate bus activity.
OS	Operating system.
page	A region in memory. The Power Architecture defines a page as a 4 KB area of memory, aligned on a 4 KB boundary or a large page size that is implementation dependent.
PBA	Pending bit array.
PBT	Push block transfer.
PCE	PCIe Configuration Environment.
PCIe	Peripheral Component Interface Express.
PEAM	Process element authority mask.
POR	Power-on reset.
POWER	Of or relating to the Power ISA or the microprocessors that implement this architecture.
Power ISA	A computer architecture that is based on the third generation of reduced instruction set computer (RISC) processors.

privileged mode	Also known as supervisor mode. The permission level of operating system instructions. The instructions are described in *PowerPC Architecture, Book III* and are required of software that accesses system-critical resources.
problem state	The permission level of user instructions. The instructions are described in *Power ISA, Books I and II* and are required of software that implements application programs.
PSL	POWER Service Layer. It is the interface logic for a coherently attached accelerator and provides two main functions: moves data between accelerator function units (AFUs) and main storage, and synchronizes the transfers with the rest of the processing units in the system.
PTE	Page table entry. A table that maps virtual addresses (VAs) to real addresses (RAs) and contains related protection parameters and other information about memory locations.
RA	Real address. An address for physical storage, which includes physical memory, local storage (LS), and memory mapped I/O registers. The maximum size of the real-address space is 2^{62} bytes.
ROM	Read-only memory.
SLB	Segment lookaside buffer. It is used to map an effective address to a virtual address.
slbia	SLB invalidate all instruction.
slbie	SLB invalidate entry instruction.
slbmte	SLB move to entry instruction.
SSE	System software environment.
SST	Storage segment table.
STABORG	Segment table origin.
STE	Segment table entry.
storage model	A CAIA-compliant accelerator implements a storage model consistent with the Power ISA.
sync	Synchronize instruction.
synchronization	The process of arranging storage operations to complete in the order of occurrence.
system software	Software that has access to the privileged modes of the architecture. System software is a reference to hypervisor and operating system software.
TAG	PSL command tag.
tag group	A group of PSL commands. Each PSL command is tagged with an n-bit tag group identifier. An AFU can use this identifier to check or wait on the completion of all queued commands in one or more tag groups.

TLB	Translation lookaside buffer. An on-chip cache that translates virtual addresses (VAs) to real addresses (RAs). A TLB caches page-table entries for the most recently accessed pages, thereby eliminating the necessity to access the page table from memory during load-store operations.
tlbie	Translation lookaside buffer invalidate entry instruction.
VA	Virtual address. An address to the virtual-memory space, which is typically much larger than the real address space and includes pages stored on disk. It is translated from an effective address by a segmentation mechanism and used by the paging mechanism to obtain the real address (RA). The maximum size of the virtual-address space is 2^{65} bytes.
VPD	Vital product data.
VPN	Virtual page number. The number of the page in virtual memory.
VSEC	Vendor-Specific Extended Configuration Capability
VSID	Virtual segment ID.
WIMG bits	Four bits in the page table, also called a page-table entry, which control the processor's accesses to cache and main storage. ‘W’ stands for write through, ‘I’ for cache inhibit, ‘M’ for memory coherence, and ‘G’ for guarded storage.
word	Four bytes.